NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

AN OBJECT-ORIENTED APPROACH TO COMPUTER ARCHITECTURE SIMULATION

by

Kevin A. Fontes

September 1991

Thesis Advisor Michel L. Nelson

Approved for public release; distribution is unlimited.

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
Unclassified

1b. RESTRICTIVE MARKINGS

2a. SECURITY CLASSIFICATION AUTHORITY

2b. DCLASSIFICATION/DOWNGRADING SCHEDULE

3. DISTRIBUTION/AVAILABILITY OF REPORT
Approved for public release; distribution is unlimited.

4. PERFORMING ORGANIZATION REPORT NUMBER(S)

5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION
Naval Postgraduate School

6b. OFFICE SYMBOL *(If Applicable)*
37

7a. NAME OF MONITORING ORGANIZATION
Naval Postgraduate School

6c. ADDRESS *(city, state, and ZIP code)*
Monterey, CA 93943-5000

7b. ADDRESS *(city, state, and ZIP code)*
Monterey, CA 93943-5000

8a. NAME OF FUNDING/SPONSORING ORGANIZATION

6b. OFFICE SYMBOL *(If Applicable)*

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

8c. ADDRESS *(city, state, and ZIP code)*

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
AN OBJECT-ORIENTED APPROACH TO COMPUTER ARCHITECTURE SIMULATION

12. PERSONAL AUTHOR(S)
FONTES, Kevin Anthony

13a. TYPE OF REPORT
Master's Thesis

13b. TIME COVERED
FROM TO

14. DATE OF REPORT *(year, month, day)*
1991 September

15. PAGE COUNT
208

16. SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17. COSATI CODES

FIELD	GROUP	SUBGROUP

18. SUBJECT TERMS *(continue on reverse if necessary and identify by block number)*
Computer Architecture, Object-Oriented Programming, Simulation, Modeling, Object-Oriented Design

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

An object-oriented approach to modeling and simulating computer architectures is presented. This approach yields a 'generic' class hierarchy that supports the simulation of basic computer microarchitecture components found in most computers. This is accomplished by concentrating on the more generic concepts of processors, memories, registers etc., rather than concentrating on a specific system. The 'generic' class hierarchy is tested by developing microarchitecture simulators for two different microarchitecture designs.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
☒ UNCLASSIFIED/UNLIMITED □ SAME AS RPT. □ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL
Michael L. Nelson

22b. TELEPHONE (Include Area Code)
(408) 646-2026

22c. OFFICE SYMBOL
CSNe

83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

An Object-Oriented Approach To Computer Architecture Simulation

by

Kevin Anthony Fontes
Lieutenant, United States Navy
B.S., California Polytechnic State University University, 1984

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

September 1991

Author: ______________________
Kevin A. Fontes

Approved by: ______________________
Michael L. Nelson, Thesis Advisor

Amr Zaky
Amr Zaky, Second Reader

Robert B. McGhee
Robert B. McGhee, Chairman
Department of Computer Science

ABSTRACT

An object-oriented approach to modeling and simulating computer architectures is presented. This approach yields a 'generic'-class hierarchy that supports the simulation of basic computer microarchitecture components found in most computers. This is accomplished by concentrating on the more generic concepts of processors, memories, registers etc., rather than concentrating on a specific system. The 'generic' class hierarchy is tested by developing microarchitecture simulators for two different microarchitecture designs.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to thank MAJ Nelson for his thoughtful suggestions and help. Also Amr Zaky for his helpful suggestions. Thanks are also in order for Lou Stevens who was able to locate an appropriate microarchitecture design to simualte in this thesis. Also thanks go out to two TGS programmers, Lynn McKaig and Mark Szpakowski for their help with Prograph coding questions.

I would also like to thank my wife, Rebecca for her proof reading and support.

I. INTRODUCTION

Designing a new computer architecture is a complex process. This process produces a complicated device, composed of many interdependent components. Since the final product is a piece of hardware, it is expensive and time-consuming to design, test, alter, and then re-test the various components.

It is difficult to directly measure the performance of new hardware as it is being designed. Performance parameters include items such as how many memory accesses occur and how many times the registers exchange data in a given time period for a given program. In order to measure these parameters directly, costly monitoring equipment must be connected directly to the hardware. The problems of monitoring and testing become even more pronounced when evaluating Very Large Scale Integration (VLSI) Hardware, as it may be impossible to connect external monitoring equipment to some of the internal components. It is also important to test the effects that each component has on all others. These effects can be both electronic and logical; only the logical effects are addressed in this thesis.

One solution to the problem of component testing and hardware evaluation is to simulate the new hardware in software. This allows separate testing of individual components as well as testing of the complete system. A complete simulation environment should model the architecture as closely as possible, yet remain as general as possible. Reusability is the main attraction of using simulation for hardware and system design.

The majority of computer architecture simulations in the past have been implemented using conventional programming languages and techniques. This thesis examines the advantages of implementing computer architecture simulation using an object-oriented (OO) approach. Encapsulation of methods and variables facilitates the reusability of code, inheritance and composition allow the building of more complex components and systems. Two computer architectures have been simulated using Prograph[1] (Cox and Pietrzykowski, 1989), an object-oriented programming (OOP) language, as part of this thesis.

A. OBJECT-ORIENTED PROGRAMMING

1. Object-oriented programming terminology

It is assumed that the reader has at least some familiarity with object-oriented programming concepts. This section provides a brief introduction to object-oriented programming and terminology as applied in this thesis.

> Object-oriented programming may be summarized by the following equation: "object-oriented = objects + classes + inheritance" (Wegner, 1987, p.168)

The backbone of an object-oriented programming language is the object. *Objects* are autonomous entities that respond to messages (operations) and have a state. An object's *state* is defined by its variables (attributes), and its *operations* are defined via its methods (procedures). An object's state can only be manipulated by its methods. Therefore, to change an object's state from the outside, a message must be sent to the object telling it which method to invoke[2].

A *class* is a template from which objects may be created (Wegner, 1987, p.168). An object is an *instance* of a class. The variables making up an object's state can be divided into class variables and instance variables. A *class variable* is defined as a variable that has the same value for all instances of a particular class. An *instance variable* is one that has a unique value for each instance of a class.

[1] Prograph is a trademark of The Gunakara Sun Systems, Ltd (TGSSystems).

[2] This assumes an encapsulated approach; although most OOP languages support this idea, not all enforce it.

For example, a class **Person** could be defined which has the instance variables **name** (the unique identifier of this object), and **where** (the current location of the object), and a class variable **People** (a list of names of all the instances of this class)[3]. Three instances of class **Person** are:

(name: cannibal1,where: left, People: (caniball1, canibal2, missionary1))
(name: cannibal2, where: right, People: (caniball1, canibal2, missionary1))
(name: missioinary1, where: left, People: (caniball1, canibal2, missionary1))

Even though these instances of Person each have a different state, they all share the same operations, (e.g., walk, sleep, etc.), because they are all instances of the same class.

Inheritance allows the creation of classes of objects that are almost like another class of objects with a few incremental changes (Stefik & Bobrow, 1986, p.40). This results in a formal code sharing mechanism. A *subclass* inherits all of the variables and methods defined for its *superclass*. A simple example of inheritance is when the class **person** (instance variables: **name, where**; class variable; **people** methods: **walk, sleep**) is inherited by class **Cannibal**. Thus **Person** is the superclass and **Cannibal** is the subclass. The subclass **Cannibal** inherits all of the variables and methods of **Person** (i.e., the code is shared); the user may also add other variables and methods in defining the subclass **Cannibal**. The inheritance hierarchy can be many levels deep and complex. *Single Inheritance* is when a class can have only one superclass (usually referred to simply as inheritance). *Multiple inheritance* occurs when a class can have several superclasses.

A *composite object* (or *aggregate object*) is an object that contains other objects. That is, its variables may themselves be instances of other classes. For example, an airplane object can be defined as containing the objects wings, propeller, wheels, etc; the wing object contains flaps, covering, cables, etc.

[3] This example is taken from a classroom project involving the implementation of the missionaries and cannibals problem (CS 4114, Winter, 1990).

An *abstract class* is a class which does not have any instances (Nelson, 1990, p.6)[4], while a *concrete class* is one which does have instances. For example, if the class **Missionary** and the class **Cannibal** are both subclasses of the class **Person** and no instances of **Person** are allowed, then the class **Person** would be an abstract class. However, both the subclasses inherit all of the variables and methods defined for the class **Person**. If the classes **Missionary** and **Cannibal** do have instances, then they are concrete.

Object-oriented programming also supports encapsulation. *Encapsulation* is the strict enforcement of information-hiding (Micallef, 1988, p.13). Encapsulation refers to an object's ability to hide implementation details behind the object interface (i.e., the operations/methods defined for the object's class). When a message is sent to an object, the object performs a method which may manipulate one or more of the object's variables without the message sender being concerned with how (or even if) those variables are manipulated.

2. Prograph

All programs implemented in this thesis use the Prograph programming environment on the Macintosh[5] computer. Prograph is a pictorial, object-oriented dataflow language (Cox and Pietrzykowski, 1989, p.1). This environment was chosen because it is pictorial in nature, it easily describes class hierarchies and variables, and it is easy to use. The following discussion is not intended to be a tutorial in Prograph programming, but rather to introduce the terminology and principles of Prograph.

A simple class hierarchy represented in the Prograph environment is presented in Figure 1.1. There are three classes: **Person, Missionary** and **Cannibal**; each one

[4] Abstract classes are not enforced by Prograph, or by any OOPL that we know of (i.e., it is only a concept).

[5] Macintosh is a trademark of Apple Computer, Inc.

depicted by a hexagonal shaped icon in the *classes window*. The icon is divided in half; the left half represents the class and instance variables and the right half represents the methods associated with the class. To open the associated window the user 'double-clicks' on the desired half of the icon. The links between class **Person** and the classes **Missionary** and **Cannibal** indicate that **Person** is a superclass of the classes **Missionary** and **Cannibal**.

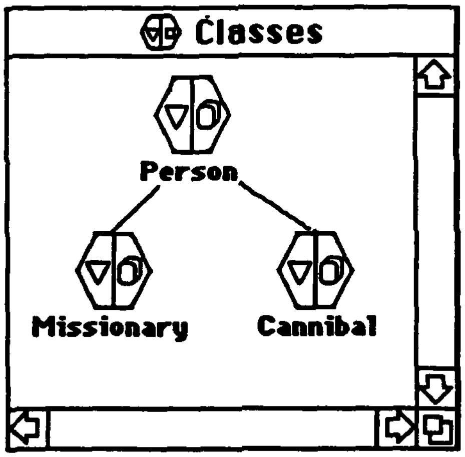


Figure 1.1 Example of Class Hierarchy and Class Attributes

The *attributes window* of the class **Person** is presented in Figure 1.2. In Prograph a class variable is referred to as a *class attribute*, and an instance variable is called an *instance attribute*. An attribute window is denoted by the inverted triangle next to the window name. The class **person,** in Figure 1.2 has two instance attributes (**name & where**) and one class attribute (**People**). Those attributes above the horizontal line represent class attributes and the attributes below the line represent instance attributes. A class attribute is represented by a small hexagonal icon similar to the class icon and an instance attribute is represented by an inverted triangle.

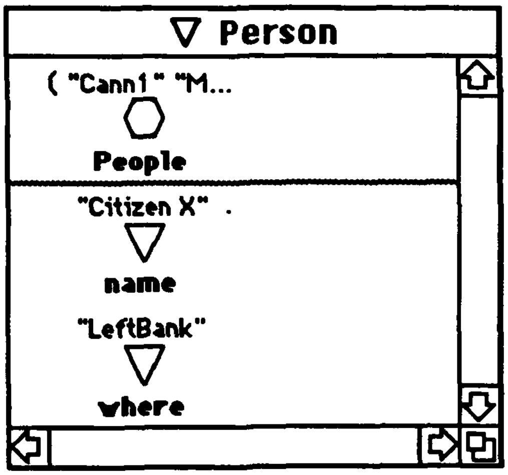


Figure 1.2 Example of Class and Instance Attributes

An example of inherited attributes is presented in Figure 1.3. An *inherited attribute* is represented by the normal attribute icon with a small downward pointing arrow. Therefore, the attributes **People**, **name**, and **where** are inherited from class **Person**. The attribute level is defined in the class Cannibal.

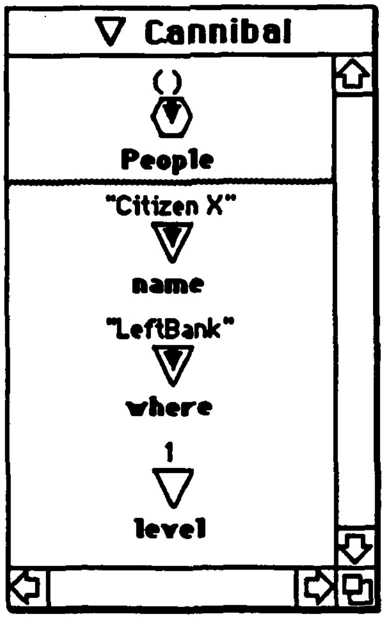


Figure 1.3 Inherited Attributes

A class' methods are represented in the *method window* as shown in Figure 1.4. The icon with a small dataflow diagram inside it next to the window name indicates that it is a method window. The six icons inside of the window depict six different methods defined for the class **Person**. Double-clicking on a method icon will open the associated methods *case window*.

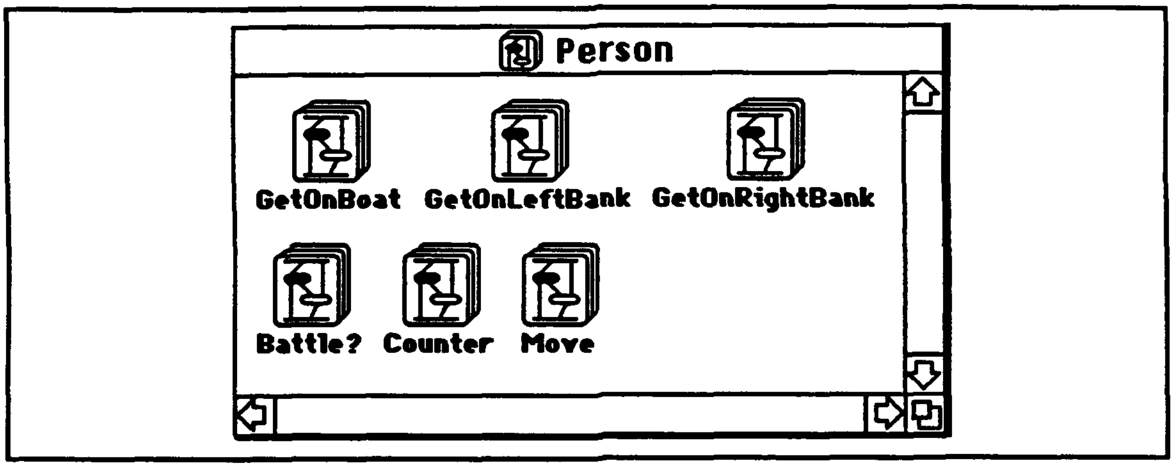


Figure 1.4 An Example of a Method Window

A case window opened from a method icon is presented in Figure 1.5 (descriptive comments are in bold type outside of the window). The window title is composed of the class name and the method name. In the example the case window shows the method **make** from the class **Cannibal.** All case windows have *input bars* and *output bars*. These bars are used to pass data into and out of the method. The numbers **1:1** in the window title indicate that this is the first case window of one case(s). When there are multiple cases of a method, all cases must have have the same number of inputs (terminals) and outputs (roots) on the input and output bars. The number of terminals or roots is defined as *arity*. Since Prograph is a dataflow language, the data flows from the top to the bottom of the case window, following the datalinks.

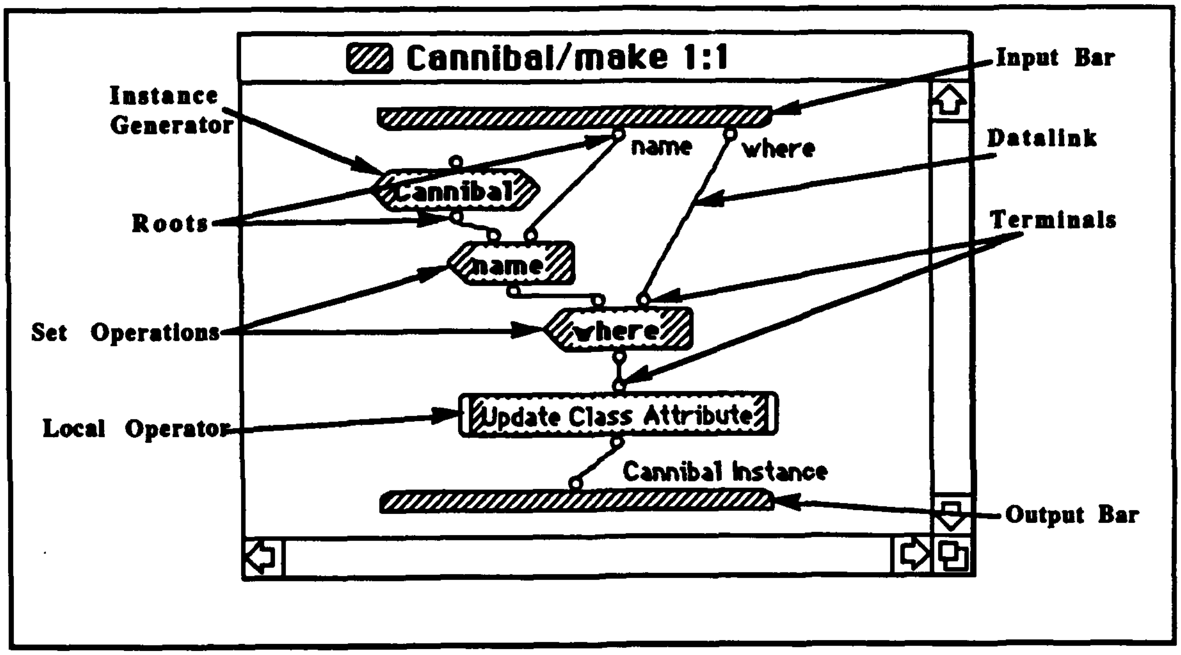


Figure 1.5 An Example of a Case Window

The shape of the icon indicates what type of operation will be performed and the name inside of the icon determines which class, attribute, or method is executed. The icon with convex left and right sides (the top leftmost operation, containing the word **Cannibal**) of Figure 1.5 is a *instance generator*. This generator is used to make an instance of the class **Cannibal** . The two operations with convex left sides below the **Cannibal** instance generator, are *set* operations. The set operation is used to set the value of an instance or class attribute. The text inside the icon determines which attribute will be set. The left terminal is used to pass the particular instance to be operated on, and the right terminal is used to input the value the instances attribute is to be set to.

The **Update Class Attribute** operation in Figure 1.5 is a *Local Operator*. Local operators are used to reduce the clutter in a case window, and are defined by the programmer. The operations inside of the local operator icon are still logically inside of the case window in which it resides; they are just grouped together in an attempt to make the window more readable. Figure 1.6 shows the contents of the local operator **Update Class Attribute** from the **Cannibal/make** window. Notice that the arity of this window is exactly the same as the arity in the associated local operator icon in the **Cannibal/make** case window.

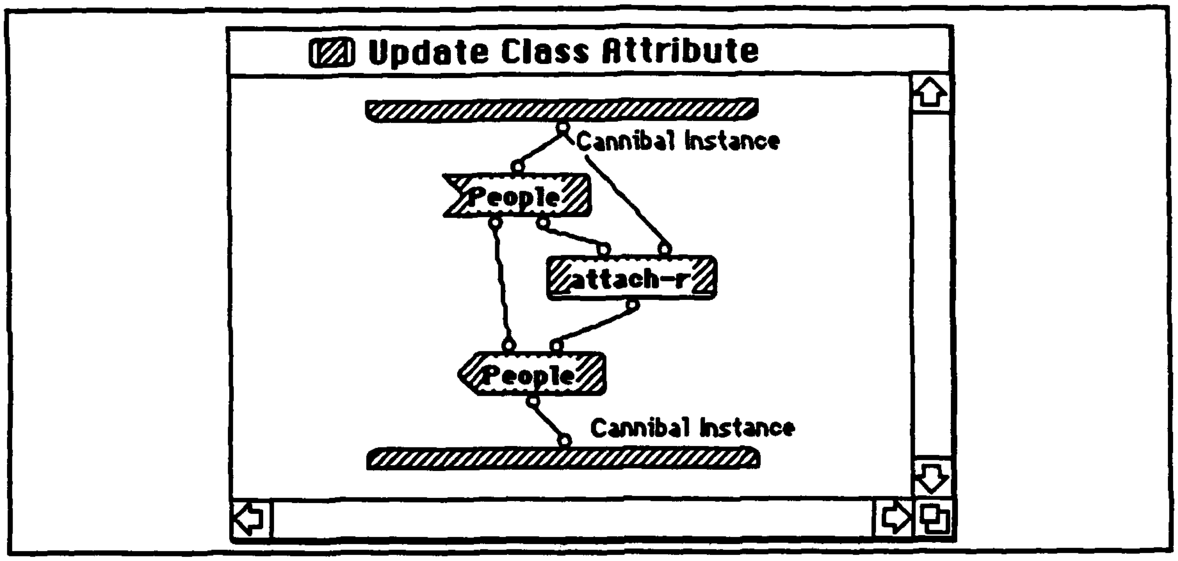


Figure 1.6 An Example of a Local Operator Window

The first operation labeled **People** (concave left side) in Figure 1.6 is a *get operation.* This get operation takes a **Cannibal** instance as input and gets the value of the **People** class attribute. The operation in Figure 1.6 labeled **attach-r** is an example of a *primitive* operation. Primitive operations are supplied by the Prograph programing environment, and are represented by an icon with a horizontal line near the base of the icon. In this case the instance of **Cannibal** coming into the window is added to the **People** class attribute (which is a list) with the **attach-r** primitive. The **People** set operation (convex left side) simply indicates that the value of the **People** class attribute has been set to the value returned by the **attach-r** operation.

There are several ways of calling a message to invoke a desired method in Prograph. Figure 1.7 gives three examples of the various representations of message calling. The text inside the operation boxes represents the message being sent. The terminals above the box are the data passed to the method, and the root leaving the box is the result of invoking the method. Regardless of the form of message passing, if the method is not found the environment searches for a method higher up in the inheritance tree. Operation A is an example of an *explicit* reference. This message tells the class **Cannibal** to invoke the **make** method. Operation B is an example of a *data-determined* reference. The message will invoke the method **make** from the class that matches the instance presented at the left input terminal. Operation C is an example of a *context-determined* reference. This type of message will invoke the method **make** from the class that matches the case window in which the operation resides.

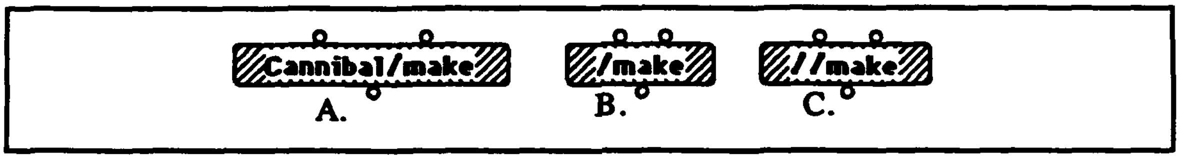


Figure 1.7 Message Passing

Progaph has two ways to handle persistent[6] data. Class attributes are used to store persistent data that is related to a specific class. This data can only be accessed via an instance of that class. *Persistents* are used to store persistent data that is not class specific. These persistents can be accessed globally. An example of the use of a persistent is presented in Figure 1.8. The persistent is represented by an oval icon (**Total** in Figure 1.8). Referring to Figure 1.8, the top persistent (with the root) is being read, and the persistent on the bottom (with the terminal) is being written to. Since Prograph is a dataflow language, program flow follows the datalinks. Using Figure 1.8, the first persistent **Total** is read, then the data is used and manipulated in the **present?** operation,

[6] Prograph uses the term persistent to describe data which is not part of a specific object.

followed by the results being stored back into the **Total** persistent. There is only one **Total** persistent, but it can be read and written as often as the programer specifies.

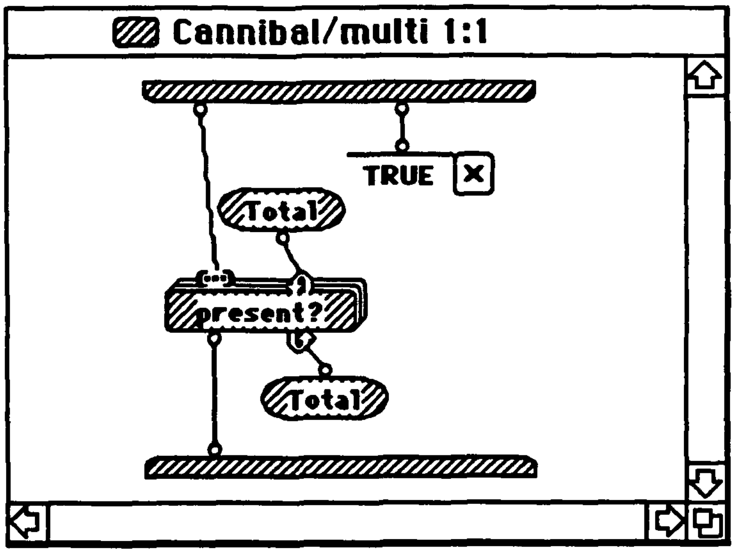


Figure 1.8 Multiplexes, Persistents and Program Control

Program flow control, an important aspect of any language, is implemented using *case control.* As previously mentioned, a case window can have multiple windows. When a case has multiple windows there must be a method to control which window will be accessed. When Prograph calls a method containing multiple case windows, it will always start execution at the first case window of the series by default. If the case window has any case control it will check that control first. Figure 1.8 also presents an example of a typical case control. The box (labeled X) in the upper right of the window is a simple case control. The X indicates that if the data that arrives at the control's terminal does not match what is in the control's box then control will transfer to the next case window of the series. There are other control symbols to perform the following: jump to next case if match, and terminate this method if match/no match.

A *multiplex* is the multiple execution of a method, which is represented as an icon that looks like a stack (i.e., several boxes, one on top of the other). The **present?** operation in Figure 1.8 is an example of a multiplex. There are several ways of controlling the number of times a particular multiplex is executed. The **present?** multiplex in Figure 1.8 has two multiplex operators. A method becomes a multiplex when one of the roots or terminals is changed from a simple root/terminal to a *list* or *loop* root/terminal. The ellipses terminal on the upper left of **present?** is a *list terminal*. This terminal will cause **present?** to execute once for each element of a list presented to its list terminal. The root/terminal pair of arrows on the right of **present?** is a *loop multiplex*. These always come in pairs. This allows the multiplex method to pass variables from one execution of the multiplex method to the next execution. The data is passed to the method on the first execution, the method uses/alters this data and passes it out of the loop multiplex where it will be passed to the input of the present multiplex method as its execution continues or it will be passed as output upon termination.

This section has presented the most basic essentials of Prograph to give the reader enough information to understand the Prograph programs used in this thesis. For more information on Prograph, please see the Prograph Reference Manual (The Gunakara Sun Systems, 1990).

B. COMPUTER ARCHITECTURE

1. Computer Microarchitecture

The *microarchitecture* level of a computer is the level that directly interacts with the actual hardware. This is the level of the computer that will be simulated in this thesis. We chose this level because it is easy to pick real components and model them as software objects. The components defined in this section are implemented as objects/classes in the following chapters.

Most computers have several common components, including: registers, memories, buses, arithmetic logic units (ALUs), and multiplexers (muxs). The computer's microacrchitecture is controlled either by a *microprogram* or by *hardwired decoding*. This thesis discusses only microarchitectures that are controlled by a microprogram. The purpose of the microprogram is to "control the machine's registers, memories, buses, ALUs, and other hardware components" (Tanenbaum, 1984, p.118). This section provides a brief introduction to the microarchitecture level components of a typical computer.

All processors (often referred to as the central processing unit or CPU) contain at least a small number of registers. *Registers* are located on the CPU chip and are used to store data. A register is characterized by the number of bits of data it can hold. For example, if a register can hold 16 bits it is referred to as a 16 bit register. The register's short access time is due to the simple circuitry required to determine the register being accessed (i. e., a relatively small number of logic gates), and also because the register is contained in the CPU chip. The CPU typically has general purpose registers which are used for storing and retrieving data encountered in instruction execution, as well as registers which have some specific function(s), such as the program counter (PC), stack pointer (SP), instruction register (IR), and accumulator (AC). All of the registers together are often referred to as a *register bank*..

A computer's *memory* is similar in construction to a register bank, except that the memory is much larger and is located some distance from the CPU (i.e., usually not on the CPU chip itself); memory is also slower than registers. Typically, a memory bank consists of many thousand locations. Like registers, memory is characterized by the number of bits of data each location can hold. It is also characterized by the number of memory locations possible. One of the reasons that memory is slower than registers is that it has many more locations which can be accessed. The larger the number of locations to access; the more digital logic required to determine the access point. The increase in the amount of access logic increases the time delay of the signals, thus producing a time delay for the desired data to be returned.

Data passed into memory is communicated via the *Memory Buffer Register (MBR)*, while the desired address of the data (location of where the data is to be stored) is loaded into the *Memory Address Register (MAR)*. A write control is activated causing the value in the MBR to be stored into the desired memory location. To read a value from a particular location in the memory the process is reversed using a read control. Thus, the memory bank interfaces with the rest of the world via an MAR, MBR, and two control signals.

A *bus* is used to transmit signals from one device to another. Physically, a bus is a collection of wires that transmit a group of signals in parallel from one location to another. Many devices can be physically connected to the same bus. The bus is considered to be an inert device. That is, if data is electrically placed on one end of the bus, the information will show up on the other end. The bus itself does not actively control the signals; rather, the bus is controlled by the equipment connected to it. If multiple devices attempt to transmit simultaneously, the values of each bit of the bus will be garbled and the data will be useless. Thus, many devices can simultaneously read data from the bus, but only one device may be transmitting at any time. Therefore, there arises a need for some central controlling device to synchronize the control of all devices writing on the bus. This will be discussed later.

Devices may receive inputs from several other devices/buses, but can only process one input at a time. This gives rise to the need for a multiplexer. A *multiplexer* is a device that has several data input lines and a single data output line. It also has several control inputs that determine which data input will be selected. The output of the multiplexer is connected to the input line of the device, and data selection is based on the multiplexer control signal. A multiplexer has 2^n data inputs and therefore has n control inputs.

The heart of the computer is the *Arithmetic Logic Unit (ALU).* The ALU typically has two inputs for data, one output for data, several control inputs, and several result output indicators. The control inputs are used to select the operation(s) to be performed on the input data. Standard ALU operations typically include: AND, OR, NOT, and ADD. The result output indicators are simple one bit signals which indicate that output of the ALU is negative, zero, overflow, etc. *Shifters* are used to shift the bits of an input to the left or right depending on the input control signals. The shifter may be placed at the output of an ALU, or it may be a part of the ALU itself.

This section has addressed the many components typically found in the *data path* side of a microarchitecture. A typical data path taken from Tanenbaum (Tanenbaum, 1984, pp.127) is shown in Figure 1.9. It includes a bank of registers, an ALU, a shifter, a Memory (including MBR and MAR) and an Amux (multiplexer).

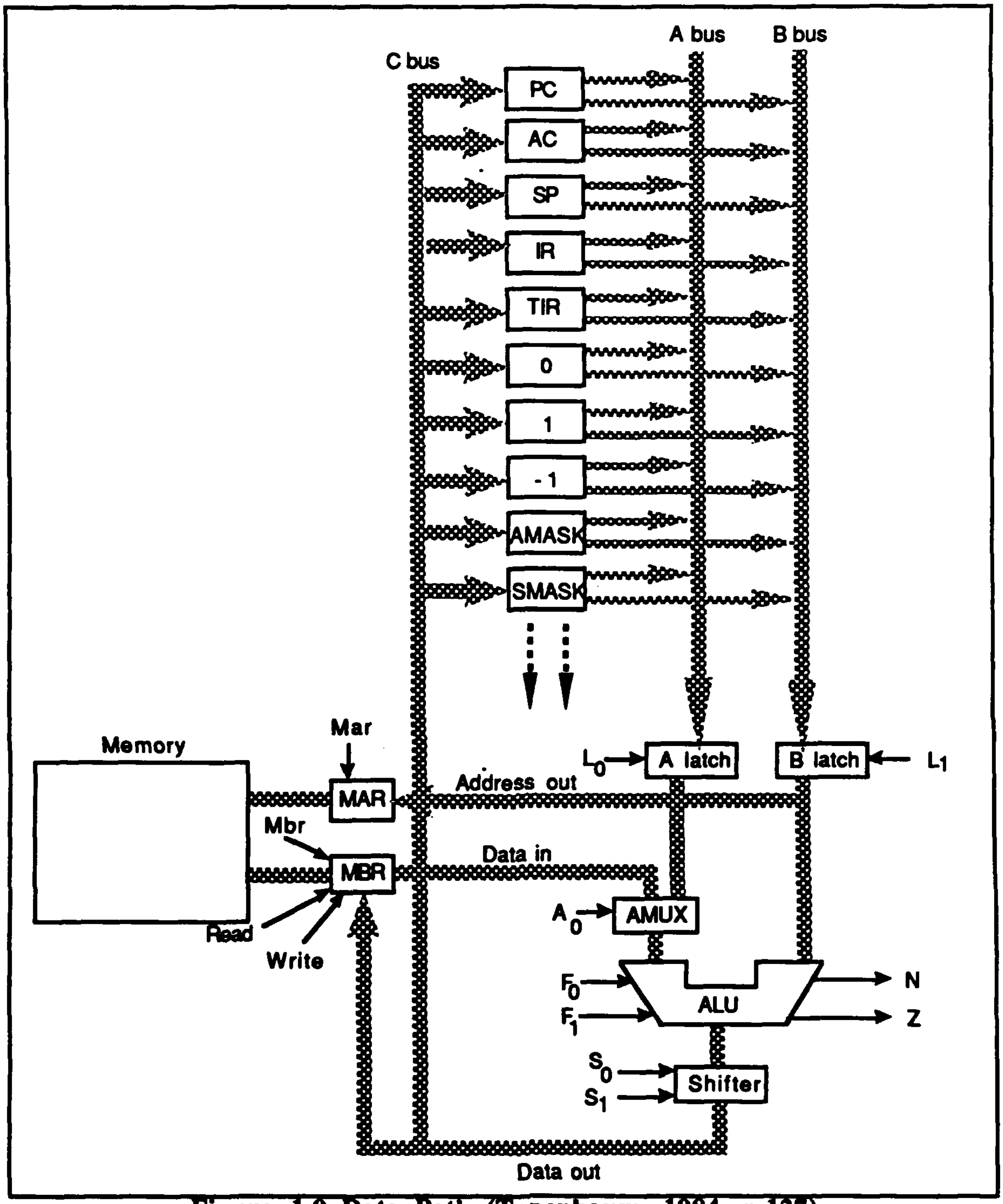


Figure 1.9 Data Path (Tanenbaum, 1984, p.127)

Figure 1.9 shows many devices along with input signals (F_0, F_1, S_0, etc), and output signals (N and Z). The input signals are generated from the *control side* of the microarchitecture. The control side is presented along with the data path in Figure 1.10. The control devices consist of: the Micro instruction register (MIR), the Control Store, and the Microprogram counter (MPC).

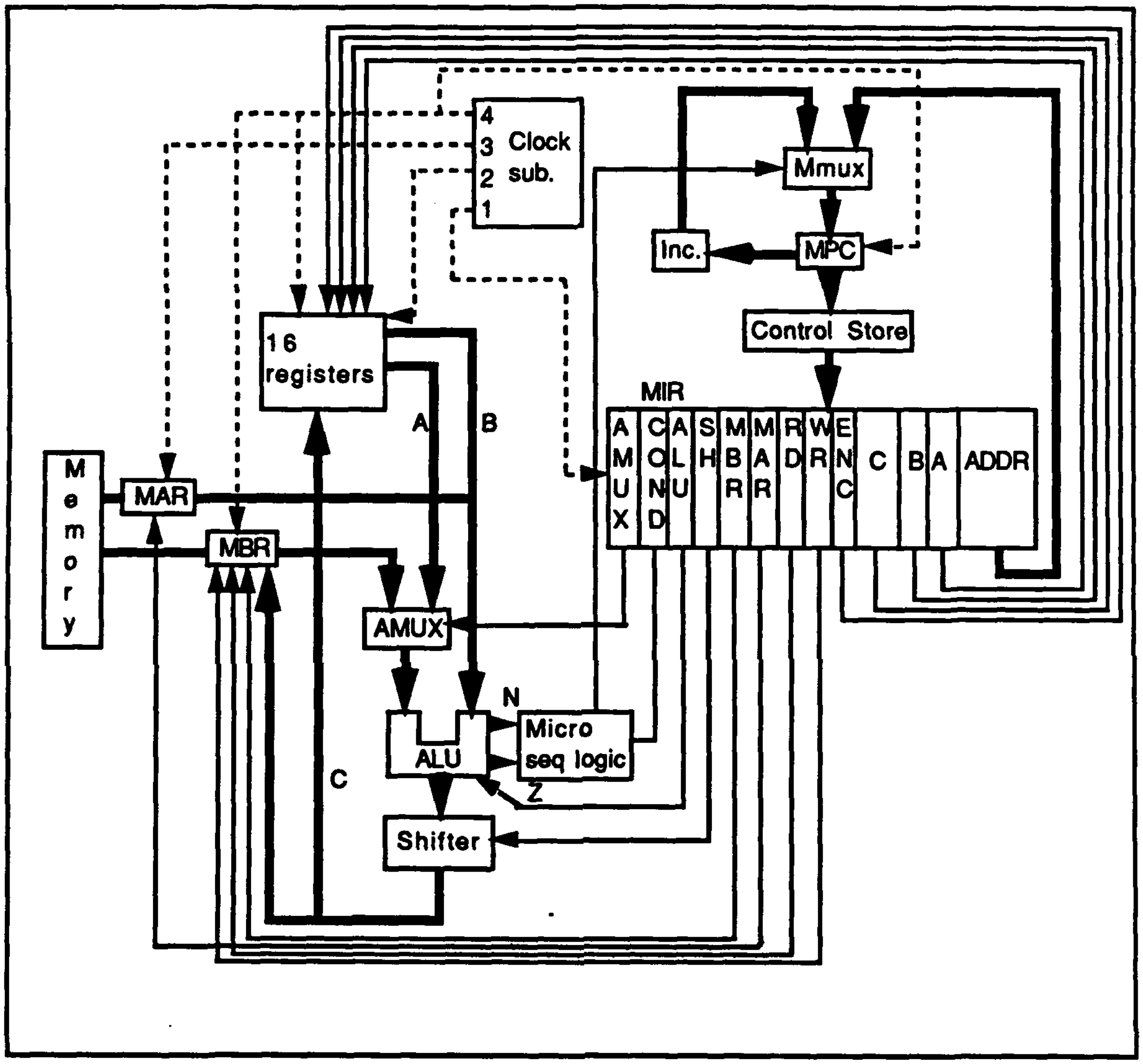


Figure 1.10 Example Microarchitecture (Tanenbaumbaum, 1984, p. 132)

The *control store* holds the entire microprogram, which consists of microinstructions. When a microinstruction is read from the control store, it is placed in the MIR. The MIR has output leads from each control field to all of the control inputs of each device in the data path side of the machine. The sequence of microinstructions is controlled by the MPC. The MPC sends its counter value to the control store, the appropriate microinstruction is retrieved from the control store and placed into the MIR, and then the MPC is incremented or changed depending on the output status signals of the ALU. This process repeats as the microprogram execution continues.

This section has introduced the many devices found in a typical computer microarchitecture. The data path consists of registers, memory, ALU, shifter, buses and multiplexers. It is controlled by the microprogram which is implemented in the MPC, control store, and MIR. Each microinstruction controls the entire microarchitecture's device control inputs. It is the continuous cycling of the microprogram which controls the fetching, decoding, and executing of higher level instructions. These components will be seen again in Chapter III where they will be simulated as software objects.

2. Simulation of Computer Architectures

Simulation of computer architecture is the use of a computer program on one computer to model the performance of another computer. Papazoglou, et. al. says that "Simulative modelling, like every other modelling approach, does offer the option that the working representation of a not yet existing system is possible" (Papazoglou, Pawlak, and Wrona, 1989, p.1). In the past, computer architectures have been modeled using classic structured programming languages. Only recently have object-oriented languages begun to be used. To model a specific architecture via a conventional language, a specific program was written to model that architecture. Whenever a simulation of a new architecture was needed, an entire new program was written to support the simulation. Papazoglou et. al. outlines the following requirements for modeling an architecture (Papazoglou, Pawlak and Wrona, 1989, p. 213):

- the model must have the same logical structure as the modelled computer system.
- it must comprise an integrated model of both the complete system hardware and software.

- it should be able to model the different parts of the system at different levels of abstraction.
- it should allow different sets of output statistics each time it is rerun with a new set of input parameters, and like any well disciplined good program it is structured, modular, reliable, efficient and extensible.

Modeling a computer architecture in an OOP language fulfills all four of these requirements, and particularly excels with the last two.

C. OVERVIEW

Chapter II is a detailed survey of the literature pertaining to previous work completed in the areas of computer modeling and object-oriented design. Chapter III is a detailed discussion of the implementation of the computer programs (developed in Prograph) simulating various computer architectures (the actual code is included in appendices C and D). Chapter IV presents the conclusions of this research effort, along with recommendations for future research and a summary of the thesis.

II. REVIEW OF THE LITERATURE

Object-oriented simulation of computer architectures is still in its infancy. The majority of the effort is in the simulation of multiprocessor systems. Most of the simulator systems that were reviewed have a very simple text based user interface; only one group is working on a simulator in which the main concern is the user interface and its ease of use. Only one group was found to be interested in hardware simulation for classroom instruction purposes. Some groups emphasize the usefulness of the object-oriented approach for the reuse of code.

To date, there is no system that incorporates reusable objects, has an intuitive user interface, was developed with commercial software, and runs on a commercially available microcomputer. Of course, a system that meets these objectives would also be economical enough to be available for classroom use. This is because most systems are not being built using commercially available software. This section discusses the various research in the field of architecture simulation, with an emphasis on object-oriented approaches.

A. SIMULATION OF COMPUTER ARCHITECTURES

A system developed at Acadia University uses a Pascal-like Register Transfer Level Language (RTLL) that operates on microcomputers (Tomek, 1985). This system was designed for the instruction of computer organization and architecture. They point out there is currently very little educational software in this field. Their package allows the user to "write descriptions of simpler CPU's, controllers and similar devices and experiment with their operation" (Tomek, 1985, p.493). The package consists of the following modules: RTLL description editor, RTLL simulator, Screen layout generator, Memory file generator, System description generator, and Organization descriptor.

The RTLL description editor is a syntax-directed editor used to develop an RTLL program. The RTLL simulator executes the program developed by the RTLL description editor. The Screen layout generator "allows the user to specify the format in which the results of the simulation are to appear on the screen" (Tomek, 1985, p.494). The memory file generator allows manipulation and loading of the memories' contents. The system description generator specifies CPU interfaces with other components. Finally, the organization descriptor is used to specify the CPU organization and timing constraints. Unfortunately they do not describe the methodology used in the development of the system or the programming language used in its implementation.

Object-oriented design has been applied to multimicrocomputer hardware and software simulation in the development of the MUDS system (Papazoglou, Pawlak, and Wrona, 1989). "MUDS constitutes an extension of the SIMULA language, and has been designed for developing prototypes of distributed software and for appropriately simulating the extension of these software prototypes on a model hardware" (Papazoglou, Pawlak, and Wrona, 1989, p.215). Thus the MUDS system is used for the simulation of hardware and software for multiprocessor computers. They introduce the design methodology used in the development of MUDS.

The basis of MUDS rests on the development of classes used to represent hardware and software. These classes are chosen such that "the structure of a designed microcomputer system may be extended with an arbitrary number of instances of the classes being modelled" (Papazoglou, Pawlak and Wrona, 1989, p.216). They emphasize that a powerful simulator can be attained with an appropriate object-oriented language, but it is essential that a proper implementation of object-oriented design techniques be used. The authors also show that the advantages of an object-oriented language (data hiding, abstraction, classes as templates, etc.) allow for a better representation of the hardware/software being modeled.

Modeling of a system can occur at various levels, For example, the microarchitecture level, macroarchitecture level, etc. The Virtural Stack Machine, a programming system for generating and interpreting code for von-Neumann machines is an example of one such level or several levels (Frei, 1989). A virtual stack machine (VSM) simulator is used to perform "VLSI netlist logic design rule checks" (Frei, 1989, p.5/1). This relatively simple architecture, modeled at the macro level, consists of three major elements: a central processing unit, a data stack, and a direct access data memory. The instruction set for the central processor is implemented as methods in one of the classes in the hierarchy. This research concluded, as with other similar research, that a class library is needed for the many objects, and that object-oriented design techniques greatly reduce the coding effort.

Nearly all hardware simulators have a very simple user interface. Only one of the systems reviewed considered the user interface as an essential facet of the simulation model. A simulator has been developed that is used for the writing and debugging of microprograms for hardware under design (Sugimoto, Abe, Kuroda, and Katou, 1988). This system was developed in a LISP-based object-oriented language, VEGAMS, which was also developed by the authors. The user interface presents a bus and component structure which graphically represents the actual hardware. This graphical representation allows the simultaneous display of the bus, register, and various other components as the simulation progresses. Having all the pertinent data available on the screen allows the microprogram developer to quickly realize mistakes and correct them immediately. Another feature is the representation of data by color coding. The microprogram is displayed in an assembly language format and an editor is provided for the system. This allows the user to alter the microprogram and reenter it into the control store while remaining in a single application.

One of the system's major features is its ability to stop at a breakpoint and roll-back the execution to some earlier time. This allows the user to examine the state variables at that time. The user can alter any variables and change the microprogram and continue execution from that point. The roll-back capability is implemented by keeping a complex linking of objects over time. Thus, to roll-back to a certain cycle number, the appropriate pointer is referenced and its data is loaded.

Although some portions of code are hardware specific, a significant portion is common to almost all computer architectures. It is claimed that developing simulators using an object-oriented language lead to approximately 62% reusability of code for other hardware simulations. As with other object-oriented applications, the design of the class hierarchy is crucial to the extensibility of the code. (Sugimoto, Abe, Kuroda, & Katou, 1988, p.55)

Simulation efforts are not limited to only object-oriented or classical languages. The simulation of concurrent real-time systems using the object-based language Ada has also been investigated (Mulcare, 1990). In the design of real-time systems, "superficial design descriptions" (Mulcare, 1990, p.184) are performed prior to attempted implementation. Of course, this leads to problems that appear during construction of the system. A formal design process followed by a comprehensive simulation of the system is easily attainable using Ada task types to model the various processes involved. The firing of a Pr-T net transition "may correspond to a task entry call" (Mulcare, 1990, p.186). Through the use of Ada generic packages, any number of various architecture components (tasks) may be instantiated. Their design methodology is described through the example design of a simple bus with interacting components which consists of specification, then Pr-T net modeling followed by Ada task/package coding. The results of the experiment concluded that very little effort was required to model the system. Also, any components modeled can become a portion of a growing reusable library of components. The author emphasizes that "the Pr-T net served to focus the entire simulation development" (Mulcare, 1990, p.189).

B. OBJECT-ORIENTED DESIGN

There are many textbooks and papers emerging on the subject of object-oriented design methodologies. Unfortunately, "To date, there is no design methodology that is universally accepted by the object-oriented community" (de Paula & Nelson, 1991, p.203). After reviewing various textbooks and papers, the design methodology proposed by de Paula and Nelson was selected (because of its ease of use) for development of systems in this thesis. The major steps of their design methodology is presented below (de Paula & Nelson, 1991, pp. 204-205):

(1) Identification of the objects and classes.
  (a) Initial definition of the objects and classes.
  (b) Analysis of the object's variables.
  (c) Analysis of the object's methods.

(2) Refinement of the objects and classes.
  (a) Addition of necessary information.
  (b) Elimination of redundant information.
  (c) Determination of class and instance variables.
  (d) Identification of composite objects.

(3) Organization of the classes into a hierarchy.
  (a) Analysis of the implementation language.
  (b) Construction of the hierarchies.
  (c) Review of the classes' variables/methods.

The power of OOP lies in its natural ability to closely map the system to the actual environment the programmer is trying to model. The first step (Identification of the objects and classes) of the procedure identifies the objects and classes necessary to build the desired model. Classes and objects can initially be derived from the problem domain, from sources such as the the following: Tängible things (cars, houses), Roles (pilot, programmer), Events (birthday, graduation), Interactions (meeting), People (manager, bricklayer), Places (Areas set aside for people or things), Things (Physical objects that are tangible), Organizations, Concepts, and Events (de Paula & Nelson, 1991).

The next step in the design process is "Refinement of the objects and classes". This step examines the methods and variables defined in the previous section. This step outlines a procedure designed to simplify the classes in preparation of building a class hierarchy.

The final step (Organization of The Classes Into a Hierarchy) of the design process organizes the classes defined in the previous steps into a class hierarchy. Class hierarchy organization has some dependency on the particular language used to implement the application.

The most important guideline de Paula and Nelson give for construction of a hierarchy is to "factor common methods as high as possible" (de Paula & Nelson, 1991, p.207). This allows the common attributes (methods and variables) of a class to be shared by all the subclasses via inheritance. If two or more classes have attributes in common but share no common superclass, an abstract class can be created as a superclass to allow these common attributes to be inherited.

de Paula and Nelson point out that when reviewing the classes' variables and methods it is necessary to look for classes that inherit unwanted variables and methods from their superclasses (de Paula & Nelson, 1991, p.6). If the inheritance of unwanted variables/methods cannot be removed, then some of the classes may have to be modified.

There is no standard format for representing class hierarchies. A simple method for specifying a class definition in a language-independent manner is given by Nelson (Nelson, 1990, p.3) as presented in Figure 2.1. This format is used for representing classes in the text of this thesis.

```
Class <class_name>
 Superclasses:          <superclass_1>, <superclass_2>, ...
 Class Variables:       <class_var_1>, <class_var_2>, ...
 Instance Variables:    <inst_var_1>, <inst_var_2>, ...
 Methods:               <method_name_1>, <method_name_2>, ...
```

Figure 2.1 Class Definition

C. CONCLUSIONS

This chapter introduced the various research in the area of computer architecture simulation. This chapter also introduced the basic concepts of object-oriented design, and described the methodology used in this work. It showed that several of the researchers are interested in developing class libraries to model computer hardware objects. Most of the researchers pointed out that the design of the class hierarchy is the crucial portion of the design of any simulator. I feel that the research outlined in this chapter demonstrates a need for a system with the following features: incorporation of reusable objects, an intuitive user interface, and development using commercial software on a commercial microcomputer. It should also be afordable for education purposes.

III. SOLUTION

Chapter I introduced the concepts necessary to understand OOP, Prograph, and the basic components of a computer microarchitecture. This chapter begins by discussing the construction of a 'generic' microarchitecture class hierarchy and the objects necessary to simulate a typical computer microarchitecture. This chapter also outlines the class hierarchy and program construction for the implementation of two different microarchitecture simulators.

A. DESIGNING A 'GENERIC' MICROARCHITECTURE CLASS HIERARCHY

This thesis uses the object-oriented design methodology presented in de Paula and Nelson's paper for the construction of class hierarchies (de Paula & Nelson, 1991). Previously discussed in Chapter II, the following is an outline of their design methodology (de Paula & Nelson, 1991, p.2):

(1) Identification of the objects and classes.
- (a) Initial definition of the objects and classes.
- (b) Analysis of the object's variables.
- (c) Analysis of the object's methods.

(2) Refinement of the objects and classes.
- (a) Addition of necessary information.
- (b) Elimination of redundant information.
- (c) Determination of class and instance variables.
- (d) Identification of composite objects.

(3) Organization of the classes into a hierarchy.
- (a) Analysis of the implementation language.
- (b) Construction of the hierarchies.
- (c) Review of the classes' variables/methods.

This methodology can be easily applied to the problem of designing the objects and classes of a computer microarchitecture.

1. Identification of the Objects and Classes

a. Initial definition of the objects and classes

In Chapter I, the components of computer microarchitectures that are common to most computers were introduced. These include: register, memory, ALU, muxs MAR, MBR, shifter, MIR, control store, and MPC. These components can be thought of as the initial set of classes.

Next an initial definition of the variables and methods associated with each of these classes must be specified. The following is the initial set of class definitions for typical components found in a computer microarchitecture as specified above:

Class: ALU
Variables: none
Methods: and, or, not, math, zero?, positive?
Description: Represents a combinational circuit; thus, it has no state. Methods are provided that perform logical operations on a stream of input data.

Class: CONTROL STORE
Variables: varies
Methods: load, read
Description: Holds the entire microprogram. It must be loaded with the microprogram prior to any simulation execution.

Class: MAR
Variables: contents (string of bits)
Methods: mar, read, write
Description: Contains an address of a **MEMORY LOCATION** within a **MEMORY BANK.** The method **mar** accepts a control signal which determines if a value is to be stored into the MAR.

Class: MBR
Variables: contents (string of bits)
Methods: mbr, read, write
Description: Models the data interface to the memory bank. The method **mbr** accepts a control signal which determines if the data input will be written to the MBR.

Class: MEMORY BANK
Variables: contents (array of
MEMORY LOCATIONs)

Methods: initialize, load, read, write
Description: A read or write to a MEMORY BANK implies a **read** or **write** to a specific MEMORY LOCATION contained in the MEMORY BANK. The method **load** is used to load a user program or data into the MEMORY BANK.

Class: MEMORY LOCATION
Variables: contents (string of bits)
Methods: initialize, read, write
Description: Used to describe the contents of a location in a MEMORY BANK.

Class: MIR
Variables: contents (string of bits)
Methods: decode, read
Description: Contains the various control signals. The method **decode** is used to parse the register contents into required control fields.

Class: MPC
Variables: contents (string of bits)
Methods: set, increment, read, jump
Description: Models a microprogram counter.

Class: MUX
Variables: none
Methods: mux
Description: Models a combinational circuit. The output is one selection from the many inputs.

Class: REGISTER
Variables: contents (string of bits)
Methods: initialize, read, write
Description: Defines how the data is represented in a system, (i.e., how many bits represents a register/memory location).

Class: REGISTER BANK
Variables: contents(array of REGISTERs)
Methods: initialize, load, read, write
Description: Similar to a MEMORY BANK.

Class: SHIFTER
Variables: none
Methods: shift_left, shift_right, no_shift
Description: Models a combinational circuit. Methods take a binary input and perform a binary shift to the left or right.

b. Analysis Of The Object's Variables

This step looks at the variables associated with the objects defined in the previous section. de Paula and Nelson recommend looking for the following (de Paula & Nelson, 1991, p.3):

1) variables that are common to groups of objects (classes)
2) variables having the same value for all objects of a class
3) variables that can be calculated or derived from other variables
4) variables that can be decomposed into more elementary variables
5) variables defined for only a single class

Applying de Paula and Nelson's guidelines to the previously described classes, we determine the following: The classes, **REGISTER, MAR, MBR, MIR,** and **MPC,** all share a common variable **contents** (guideline #1 above). However, for this application the value of **contents** for **MPC** is an integer value. Thus, the variable **contents** of the class **MPC** cannot be treated the same as the variable **contents** of the classes **REGISTER, MAR, MBR,** and **MIR.** Yet, the MPC class still requires a read method like the previously mentioned group of classes. The classes **CONTROL STORE, MEMORY BANK,** and **REGISTER BANK** also share the variable name **contents** but in this context **contents** refers to an array of the respective location types (i.e., **MEMORY LOCATIONS** for **MEMORY BANK,** etc.). Further observation of the class descriptions show no variables to which guidelines #2, #3, #4, or #5 may be applied.

c. *Analysis of the Object's Methods*

This step looks at the methods associated with the objects defined in the previous section. The following points should be considered (de Paula & Nelson, 1991, pp. 3-4):

1) Look for methods that are common to several classes.

2) Every concrete class should have, as a minimum, a set of methods to create, delete, maintain, and display its instances.

3) In order to enforce encapsulation, it may also be necessary to define methods for accessing and updating each variable.

The following is a summary of significant observations made by applying the above design guidelines to each object's methods: The classes **REGISTER, MEMORY LOCATION, MAR, MBR, MIR** have the common methods **read** and **write. REGISTER** and **MEMORY LOCATION** also share the method **initialize.** The similar classes **MEMORY BANK,** and **REGISTER BANK** share the methods **initialize, load, read,** and **write.** Also, the class **CONTROL STORE** shares the methods **load** and **read** with **MEMORY BANK** and **REGISTER BANK.**

2. Refinement of the Objects and Classes

a. *Addition of Necessary Information*

The methods **BINARY READ** and **BINARY WRITE** were added to the classes **MEMORY BANK** and **REGISTER BANK.** These classes require two types of **read** and **write** methods. Due to programming concerns it is necessary to read from and write to a storage/memory location using an integer or binary input. Therefore the **BIN-read** and **BIN-write** methods were added to these classes to allow this flexibility. The binary version of the read and write methods use the corresponding integer version of these methods, by converting the binary address to its integer equivalent and calling the integer version.

Closer examination of the object **CONTROL STORE** shows that there is still a need to determine how the microprogram will be represented. Each step in a microprogram has the same type of data, which determines what control signals will be sent. The object **CONTROL STORE** should be made up of this data. This data format can be clearly represented by introducing a new class, **MICROINSTRUCTION**, which will define the various fields necessary to make up a microprogram microinstruction. Thus, **CONTROL STORE** will consist of many instances of **MICROINSTRUCTION**. This new class **MICROINSTRUCTION** also affects the class **MIR**, in that the **MIR** contains a single **MICROINSTRUCTION**.

Examination of the classes **REGISTER, MEMORY LOCATION, MAR, MBR,** and **MIR** shows that each of these classes has the instance variable **contents**. If one considers the meaning of **contents** applied to each of these classes it becomes apparent that the value of **contents** for an instance of **REGISTER, MEMORY LOCATION, MAR, MBR,** and **MIR** consists of a single n-bit value (representing the contents of a register), while the value of **contents** for an instance of **CONTROL STORE, MEMORY BANK,** and **REGISTER BANK** will consist of an array of many n-bit numbers (one for each storage location). Each one of these storage locations can be described by an instance of that class related individual storage type. This results with the instance variable **contents** containing an array of **REGISTER, MEMORY LOCATION,** or **MICROINSTRUCTION** for its corresponding store type.

At this point, a design decision was made to represent the instance variable **contents** as a list. This allows great flexibility as Prograph provides very powerful list primitives. Representing **contents** as a list allows the application program to represent storage locations with variable lengths. In the case of **REGISTER, MAR, MBR, MIR,** and **MPC**, where **contents** represents a single binary number, contents can be represented as a list of booleans, where each boolean represents a bit of the binary number. In the case of **CONTROL STORE, REGISTER BANK,** and **MEMORY BANK**, the value of **contents** represents many binary numbers so **contents** can be represented as a list of instances of of the respective location type.

b. Elimination of Redundant Information

Redundant data is simply data which is not necessary to store directly as it may always be derived from some other data. There is no redundancy in the data maintained by the various classes defined so far.

c. Determination of Class and Instance Variables

The variables in the classes discussed above have unique values for each instance of each class. This means that these variables can be represented as instance variables.

d. Identification of Composite Objects

There are no variables in the above mentioned classes that decompose into more elementary variables. Thus, there are no composite objects.

3. Organization of The Classes Into a Hierarchy

a. Analysis of the Implementation Language

The following questions should be considered (de Paula & Nelson, 1991, p.2):

1) Does the system provide single, multiple, or selective inheritance?

2) If multiple inheritance is supported, what are the conflict resolution rules?

3) Can inherited methods be redefined (overridden) in the subclasses?

4) Can inherited variables be redefined (overridden) in the subclasses?

The implementation programs supporting this thesis are implemented in Prograph, an object-oriented programming environment. Prograph supports single inheritance only, which answers question one above and makes question two not applicable. In answer to question three, Prograph allows inherited methods to be modified or redefined, allowing the superclass to keep its original definition while allowing modifications in the subclass method. Prograph also allows inherited variables to be redefined which answers question four. Therefore, the object-oriented programs for this thesis will have the following features: 1) single inheritance; 2) inherited methods can be redefined in subclasses; and 3) inherited variables can also be redefined in subclasses.

b. Construction of the Hierarchies

In section A.1.b it was determined that **REGISTER, MEMORY LOCATION, MAR, MBR, MIR,** and **MPC** have the same instance variable **contents.** These classes also share several methods. The classes **MAR, MBR, MIR,** and **MPC** represent specific applications of the more general class **REGISTER** which allows these classes to be subclasses of the class **REGISTER.** Since the classes **REGISTER** and **MEMORY LOCATION** share a common instance variable, and several methods, but share no common superclass, it was decided to add the abstract class **STORAGE LOCATION.** With the class **STORAGE LOCATION** defined, the methods **initialize, read** and **write** can be moved up to the **STORAGE LOCATION** class.

Further examination of the above class descriptions also reveals that the classes **CONTROL STORE, MEMORY BANK,** and **REGISTER BANK** share a common instance variable and many common methods, but no common superclass. Thus, the abstract class **STORAGE BANK** was defined as a superclass for these classes. The methods **initialize, read write BIN-read, BIN-write,** and **load** can then be moved up to the **STORAGE BANK** class.

The remaining classes **ALU, MICROINSTRUCTION, MUX,** and **SHIFTER** have no commonalities with any other class, and are therefore unrelated to other classes by inheritance. Figure 3.1 presents the class hierarchy that results from the above discussion.

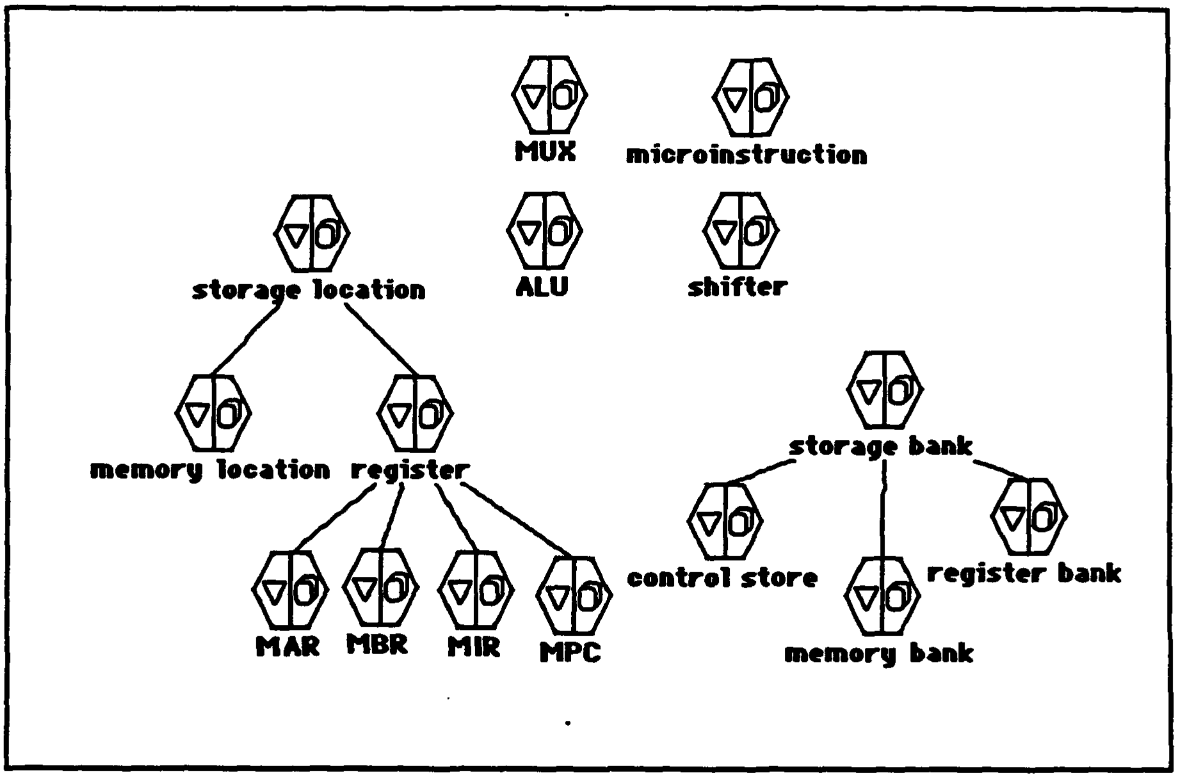


Figure 3.1 'Generic' Class Hierarchy

We can now present a revised list of the 'generic' computer microarchitecture class definitions:

Class: ALU	
Superclass:	none
Variables:	none
Methods:	and, or, not, math, zero?, positive?

Class: CONTROL STORE
Superclass:
Variables: contents:
(array of MICROINSTRUCTIONs)
Methods: none

Class: MAR
Superclass: REGISTER
Variables: none
Methods: mar

Class: MBR
Superclass: REGISTER
Variables: none
Methods: mbr

Class: MEMORY BANK
Superclass: STORAGE LOCATION
Variables: contents: array of MEMORY LOCATIONs
Methods: none

Class: MEMORY LOCATION
Superclass: STORAGE LOCATION
Variables: none
Methods: none

Class: MICROINSTRUCTION
Superclass: none
Variables: instruction (string of bits)
Methods: none

Class: MIR
Superclass: REGISTER
Variables: none
Methods: decode

Class: MPC
Superclass: REGISTER
Variables: none
Methods: set, increment, jump

Class: MUX
Superclass:
Variables: none
Methods: mux

Class: REGISTER
Superclass: STORAGE LOCATION
Variables: none
Methods: none

Class: REGISTER BANK
Superclass: STORAGE BANK
Variables: contents: array of REGISTERs
Methods: none

Class: SHIFTER
Superclass: none
Variables: none
Methods: shift left, shift right, no shift

Class: STORAGE BANK
Superclass: none
Variables: contents: array of STORAGE LOCATIONs
Methods: initialize, load, read, write, binary read, binary write

Class: STORAGE LOCATION
Superclass: none
Variables: contents: string of bits
Methods: initialize, read, write

c. *Review of the classes' variables/methods*

The constructed class hierarchy does not introduce any unnecessary variables or methods in any class. We believe that it provides the basis for an accurate model of the real world situation.

B. IMPLEMENTATION OF TANENBAUM'S MICROARCHITECTURE

With the class hierarchy of a general microarchitecture designed, it is now relatively easy to implement the design for a specific microarchitecture. A simple microarchitecture presented by Tanenbaum (Tanenbaum, 1984, pp. 126-149) can be modeled using the classes presented in section A.3.b. A complete block diagram of his microarchitecture design is reproduced in Figure 3.2 below:

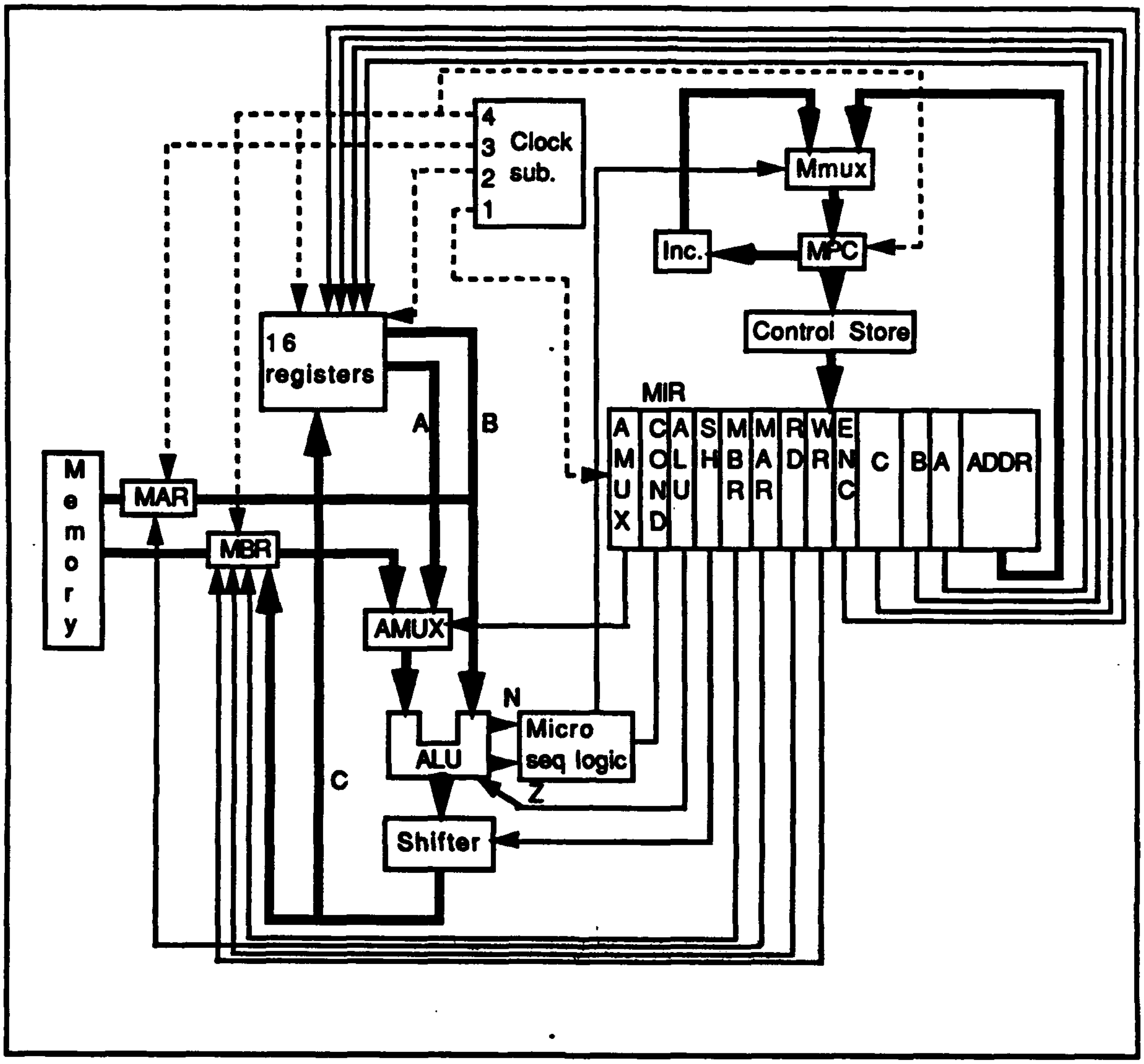


Figure 3.2 Tanenbaum's Microarchitecture (Tanenbaum, 1984, p. 132)

1. Operation of the Tanenbaum Microarchitecture

This section is a summary of the design and operation of Tanenbaum's example microarchitecture; for more detail on this design refer to (Tanenbaum, 1984, pp. 126-149). This microarchitecture design is divided into two main subsections; the datapath and the control path. The left side of Figure 3.2 is the data path and the right side is the control path. The data path side of this design consists of a 16 location register bank, AMUX, ALU, SHIFTER, MAR, MBR, and memory. The bus width of the datapath is 16 bits, all registers and memory locations are also 16 bits. Some of the register locations are for general use and others are for specific use; a summary of the purpose of the various register locations is summarized in Figure 3.3.

Location	Purpose	Symbol
00	Program Counter	PC
01	Accumulator	AC
02	Stack Pointer	SP
03	Instruction Register	IR
04	Temporary Instruction Register	TIR
05	Zero	0
06	+1	1
07	-1	-1
08	AMASK (address mask)	0FFF (hex)
09	SMASK (stack mask)	00FF (hex)
10-15	General Purpose Registers	

Figure 3.3 Microarchitecture Register Uses

Two of the registers can be read simultaneously and placed on the A and B buses. The A bus signal is fed into the AMUX along with a signal from the MBR. Depending on the control signal (0-A bus, 1-MBR) to the AMUX (a simple two input multiplexer), one of the two inputs will be passed on to the left input of the ALU. The value on the B bus is fed directly into the right input of the ALU. The value on the B bus is also routed to the input of the MAR, if the control signal to the MAR is TRUE the value of the B bus will be read into the MAR.

Two control signals, F_0 and F_1, cause the ALU to perform one of the following operations: A+B, A AND B, A, ~A (where A and B represent the data on the respective buses). The ALU generates one data output and two control outputs. The data output feeds to the input of the shifter. The two control output signals are Z (true if data result is zero) and N (true if the data result is negative). These two signals feed into the micro sequencing logic.

The shifter shifts the input data one bit to the left or right, or it can pass the data through to the C bus without alteration. The shifter has two control signals as input, S_0, and S_1. The output of the shifter is placed on the C bus and can be fed to the register bank and the MBR. The value of the C bus will be loaded into the desired location in the register bank if the ENC control is TRUE. The value of the C bus will be loaded into the MBR if the MBR signal is also TRUE.

The MAR and MBR in this design can be considered to be the interface between the memory and the CPU. When the MBR receives a READ signal of TRUE it will read the contents of the memory location pointed to by the MAR and place that location's value into the MBR. When the MBR receives a WRITE signal of TRUE it will place the contents of the MBR into the memory location pointed to by the value of the MAR. As discussed in the introduction, the access speed of memory is usually slower than the access speed of the CPU's registers. In this design it takes two complete machine cycles to read from or write to memory. This means that register access is twice as fast as memory access.

This architecture uses a microcoded program to control its components. The microprogram is stored in the Control Store. At the beginning of each clock cycle, the contents of a location (pointed to by the MPC) of the control store is loaded into the MIR. The MIR is divided into various fields, each field holding a control signal for a specific component. These control signals are routed to the various hardware components in the microarchitecture. As soon as the desired microinstruction is loaded into the MIR, the signals are routed to these components for the entire clock cycle. The status of the microprogram is maintained by the MPC. After the MPC is read its value is incremented and the result is presented to the Mmux. Two of the fields of the microinstruction are the

ADDR and the COND fields. The value of the ADDR field is presented to the input of the Mmux. The Mmux chooses between the ADDR and increment inputs for an output. The selection depends on the output of the micro sequencing logic (based on the outputs of N and Z) and the value of the COND read from the MIR. The COND code is summarized in Figure 3.4 (Tanenbaum, 1984, p.134):

0 = Do not jump; next microinstruction is taken from MPC + 1
1 = Jump to ADDR if N = 1
2 = Jump to ADDR if Z = 1
3 = Jump to ADDR unconditionally

Figure 3.4 COND Code Definitions

This result determines if the next microinstruction executed will be the next instruction in the control program's sequence or a jump to some other portion in the microprogram.

Each clock cycle is divided up into four subcycles. The following events occur during these subcycles (Tanenbaum, 1984, p.131):

1. Load the next microinstruction into the MIR, send control signals to various components.
2. Gate registers onto the A and B buses, increment the MPC.
3. Allow ALU and shifter time to produce stable outputs and load MAR if required.
4. Store the C bus into the desired register location if desired and load the MBR if required.

2. Design of The Class Hierarchy

The design of a class hierarchy to implement a simulation program for Tanenbaum's microarchitecture begins with the general microarchitecture classes outlined in section A.3.b. and building from them into a full Macintosh application. Appendix C contains the complete Prograph source code listing for the simulation of Tanenbaum's microarchitecture. A design decision was made to allow the microarchitecture to simulate memories and registers with a variable bit width. This allows for quicker test runs and also allows the simulator to model memories of various sizes.

a. Review and Modification of the General Class Hierarchy

No modifications (from section A.3.b) are required to the following classes: **ALU, MAR, MBR, MIR, MUX, MEMORY LOCATION, MICROINSTRUCTION, REGISTER, REGISTER BANK, SHIFTER, STORAGE BANK,** and **STORAGE LOCATION.** The following classes were modified (The revised 'generic' class descriptions are located in Appendix A):

Class: CONTROL STORE
modification: Added a load method that invokes the inherited load method from storage bank to assign appropriate variable values.

Class: MEMORY BANK
modification: 1) Added a load method that invokes the inherited load method from storage bank to assign appropriate variable values. 2) Overshadowed inherited methods read and write (from STORAGE BANK) to support read/write using MAR/MBR interaction with the memory.

Class: REGISTER BANK
modification: Added a load method that invokes the inherited load method from storage bank to assign appropriate variable values.

Class: MPC
modification: 1) Added instance variables **cycles & counter** for simulation support. 2) Added methods **set cycles and get counter.** Set cycles is used to set the cycle counter to zero and store the desired number of clock cycles in an instance of MPC. The method get counter, passes the counter value of an instance of MPC to its calling method.

Class: MICRO SEQUENCER
Superclass: none
Variables: none
Methods: generate signal

Description: Simulates the micro sequencer component of Tanenbaum's architectue. The method **generate signal** sends a signal to the mux for controlling logic and addressing of the MPC.

The source code listing in Appendix C also includes several other classes that have not been discussed thus far. These classes include: **System, Application, Menu, Menu Item, Window, sim** and **Time**. The **time** class is supplied in a class library provided by TGS systems (The TGS systems bulletin board). This class is used to convert system time to strings for I/O purposes. The 'About Micro Simulator' menu item displays a dialog box that gives program credits and the date and time. The **sim** class is a class added to the hierarchy (inheriting variables/methods from the class **Window**) which contains methods that implement input/output for simulation purposes. The other classes are all System classes supplied with the Prograph interpreter/compiler (TGS Systems, 1990, p.109). These classes must be included with any application; they include the attributes and methods necessary to handle menus, windows, and event control for stand alone applications.

3. Design of the Micro Simulator

a. The user interface

The Micro Simulator was designed to be a stand alone Macintosh application. This means that the user interface consists of a menu bar for controlling the program and windows and dialog boxes for facilitating input/output. The Menu Bar is shown in Figure 3.5.

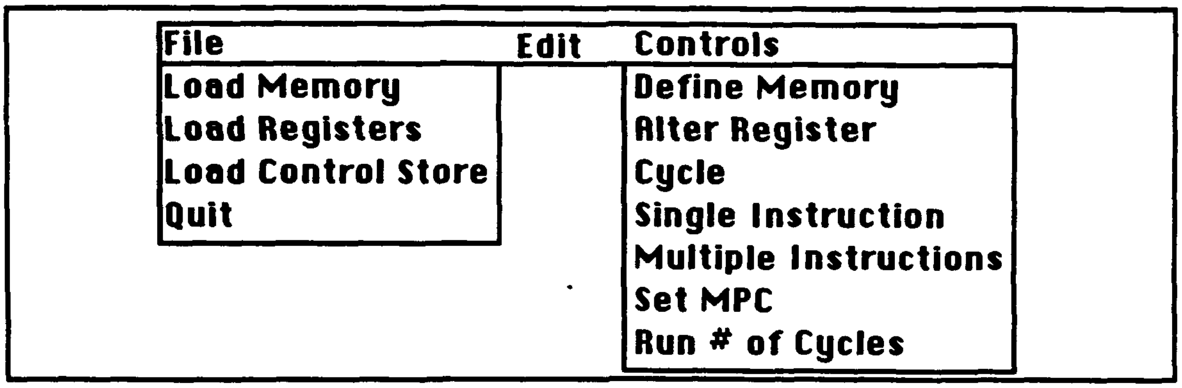
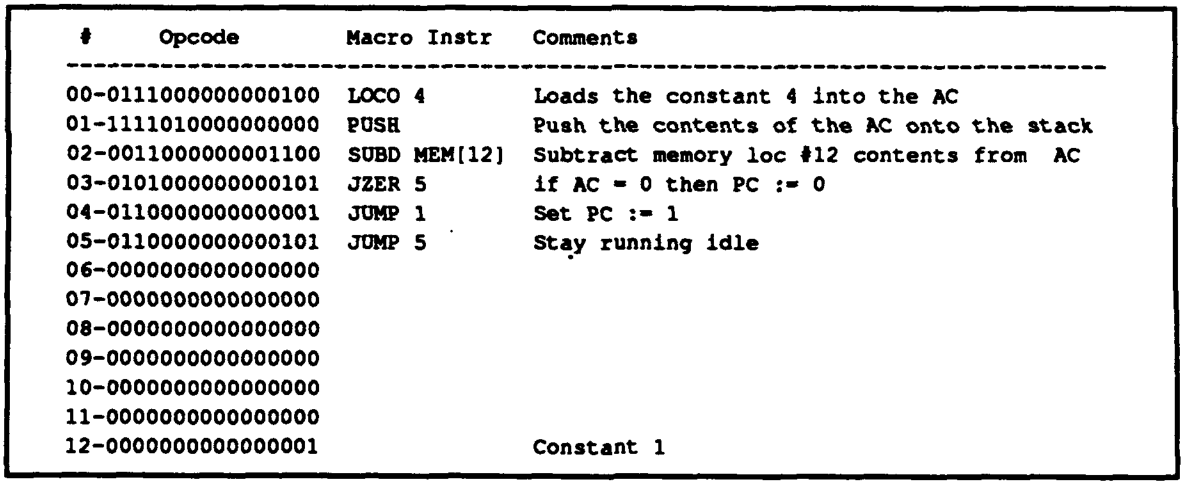


Figure 3.5 Micro Simulator's Menu Bar

Referring to Figure 3.5, the first menu item under the File menu is **Load Memory**. This selection is used to load a text file that represents the initial memory map into the memory bank. A sample program that can be loaded into memory using the **Load Memory** selection is presented in Figure 3.6.



```
#      Opcode          Macro Instr   Comments
-----------------------------------------------------------------------------
00-0111000000000100   LOCO 4        Loads the constant 4 into the AC
01-1111010000000000   PUSH          Push the contents of the AC onto the stack
02-0011000000001100   SUBD MEM[12]  Subtract memory loc #12 contents from  AC
03-0101000000000101   JZER 5        if AC = 0 then PC := 0
04-0110000000000001   JUMP 1        Set PC := 1
05-0110000000000101   JUMP 5        Stay running idle
06-0000000000000000
07-0000000000000000
08-0000000000000000
09-0000000000000000
10-0000000000000000
11-0000000000000000
12-0000000000000001                 Constant 1
```

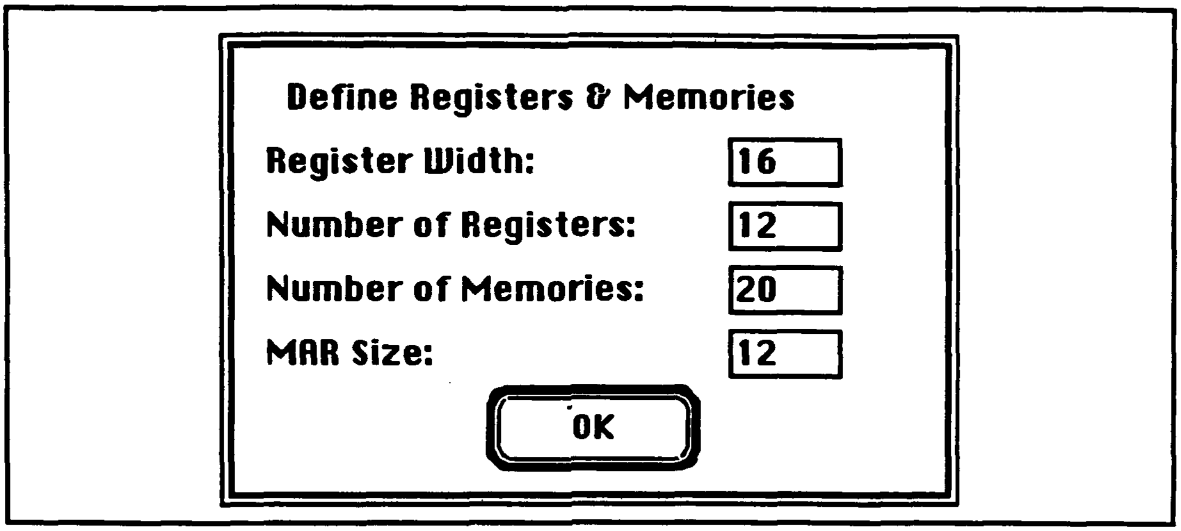
Figure 3.6 Sample Object Program Text Format

This is a simple program that loads the constant 4 into the accumulator and then pushes copies of it five times on top of the stack.The numbers preceding the dash are the desired memory location. The numbers following the dash are the instruction opcodes or memory values. These are the actual values loaded into the memory. Any characters following the opcodes are considered comments and are not loaded. The first column following the opcode is a macro description of the preceding opcode. The column following the macro description contains general comments. The text file has to be as long as the program requires, if the program contains 13 instructions then the text file must contain 13 lines. Notice that locations six through 11 have no values; that is because they are used as place holders for an actual constant value at location 12. If these place holders were not used the constant that should have been loaded at location twelve would instead be loaded at location 6.

Referring once again to the **File** menu of Figure 3.5, the **Load Registers** selection operates exactly the same as the **Load Memory** selection, except that the data is directed to the **Register** bank. The **Load Control Store** selection similarly loads a text file into the **Control Store**. The **Quit** selection is used to quit the Micro Simulator application.

The **Edit** menu selection of Figure 3.3 is a standard Macintosh menu bar selection and must be present for all Macintosh applications. For more information on this selection, refer to Inside Macintosh Volume I (Apple Computer, 1985, p.58).

The **Controls** menu of Figure 3.4 is the menu which implements the features of the Micro Simulator. The **Define Memory** selection is used to initialize the simulator's register and memory configuration. This selection causes a dialog box to be displayed as shown in Figure 3.7. The first three entries are self explanatory. They determine the width, in bits, of the memory/registers and the respective number of locations for each. **MAR** size determines the desired bit width of the MAR. This allows the simulation to limit the address space allowed for the memory interface.



Define Registers & Memories

Register Width: 16

Number of Registers: 12

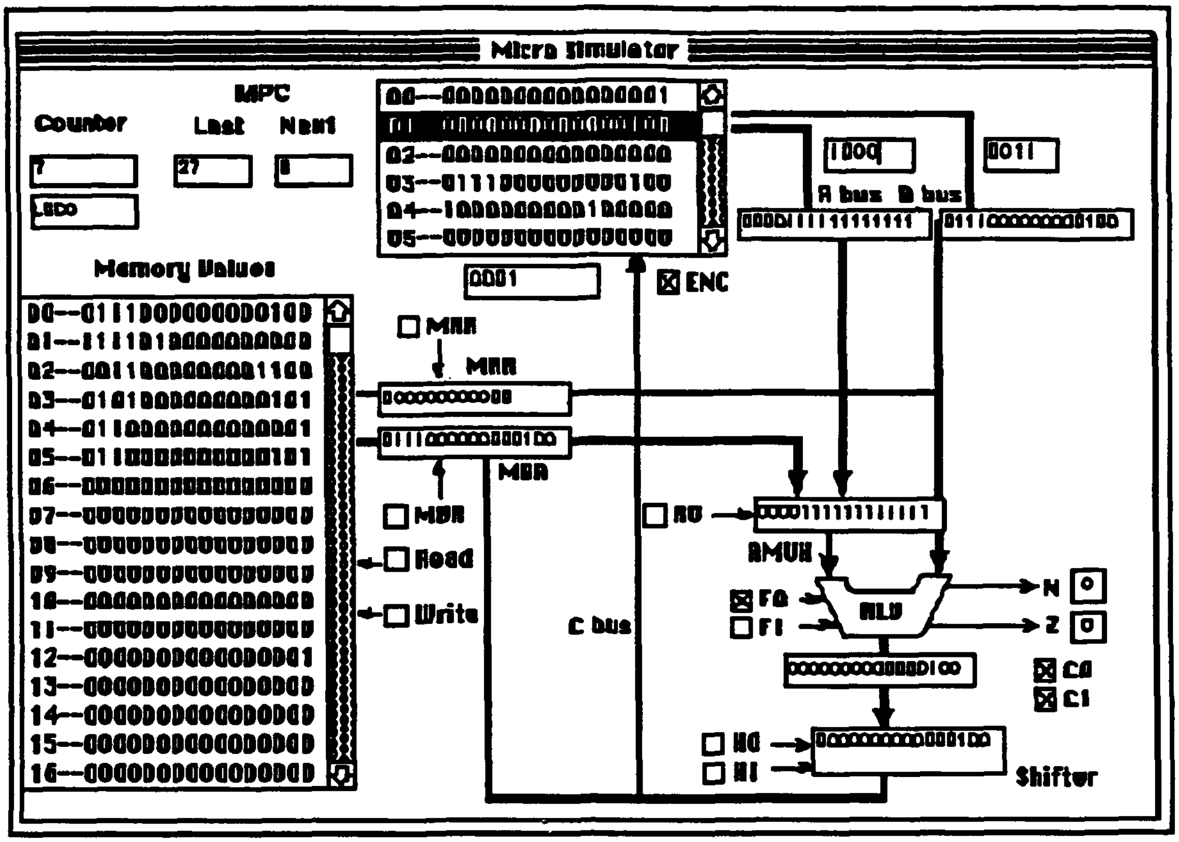
Number of Memories: 20

MAR Size: 12

Figure 3.7 The Define Memory Dialog Box

Referring once again to the Controls menu of Figure 3.4, the **Alter Register** command displays a dialog box which allows the user to input a desired register location (by number) and the new value to load into that register from the keyboard. **Cycle** causes the simulator to execute one microinstruction. **Single Instruction** causes the simulator to execute one macro instruction. The **Multiple Instructions** command displays a simple dialog box that requests the desired number of macro instructions to be executed; this causes the simulator to execute that number of macroinstructions. The **Set MPC** command displays a dialog box which allows the user to set the MPC. The **Run # of Cycles** command displays a dialog box asking for the desired number of clock cycles to be executed, then executes the desired number of clock cycles and stops.

Status of the simulator is displayed in a window (Micro Simulator) which includes a diagram of the data side of the microarchitecture along with the values of the various components. A reduced copy of the Micro Simulator window is presented in Figure 3.8. Any values displayed are for the last microinstruction executed.



Counter: 7
LSDO
Last: 27
Next: 8

00—0000000000000000001
01—0000000000000000100
02—0000000000000000000
03—0111000000000100
04—1000000000100000
05—0000000000000000

0001

☒ ENC

1000
0011

0000111111111111
0111000000000100

00—0111000000000100
01—1111010000000000
02—0011000000001100
03—0101000000000101
04—0110000000000001
05—0110000000000101
06—0000000000000000
07—0000000000000000
08—0000000000000000
09—0000000000000000
10—0000000000000000
11—0000000000000000
12—0000000000000001
13—0000000000000000
14—0000000000000000
15—0000000000000000
16—0000000000000000

□ MAR
000000000000
0111000000000100
□ MBR
□ Read
□ Write

□ A0
0000111111111111
☒ F0
□ F1
N 0
Z 0
0000000000000100
☒ C0
☒ C1
□ H0
□ H1
0000000000000100

Figure 3.8 The Micro Simulator Window

The various check boxes (ENC, MBR, MAR, etc.) represent control signals sent from the last microinstruction. An activated box ('x' inscribed in the box) indicates that the corresponding control signal is true, and an unactivated box indicates a control signal of false.

The rectangles containing text display the contents of the respective object. The boxes labeled A bus and B bus indicate the values of the respective buses. The smaller boxes above those values indicate what register locations (by binary number) were loaded on the respective buses. There are also text boxes to indicate the contents of: MAR, MBR, and C bus storage location. The text boxes associated with the ALU, AMUX, and Shifter indicate the outputs of those devices. The boxes under MPC Last and Next indicate the number of the previous microinstruction executed and the number of the next microinstruction to execute. The Counter text box indicates how many clock cycles since the last time MPC was set. The text box below the Counter box presents the mnemonic of the last complete macro instruction executed. Two scroll boxes are used to display the contents of the memory and register banks.

b. Micro Simulator Program Structure

The dataflow nature of Prograph allows for the Micro Simulator program structure to be relatively simple. As stated in section B.1, each clock cycle is divided into four subcycles. A universal method Cycle is a method that contains four local operators that each contain their respective subcycle. Figure 3.9 presents the logical dataflow modeled by the Prograph source code. This code simply gets the Micro Simulator window and then executes each subcycle sequentially. The dataflow implementation automatically forces each subcycle (like a precedence graph) to run to completion before the next subcycle can execute. Several universal methods are provided to drive the Cycle method for the following: one complete cycle, several cycles, and to completion of a single macro instruction.

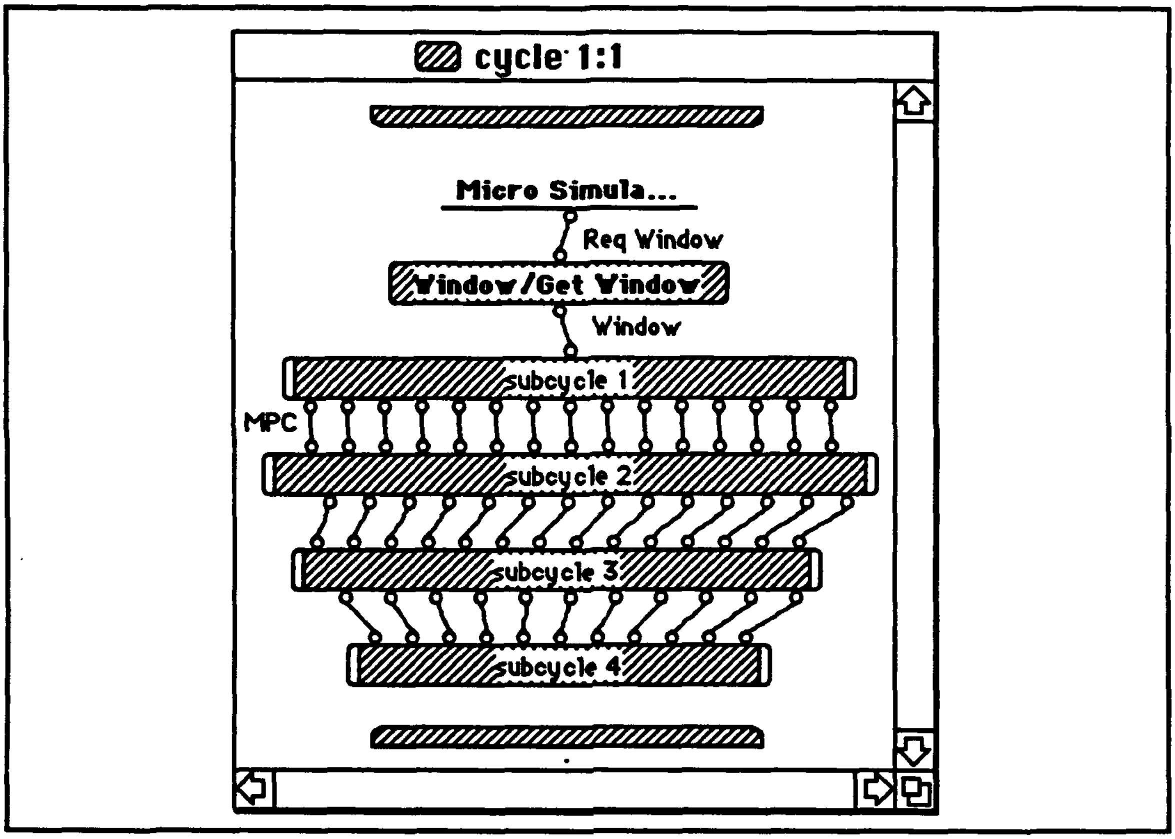
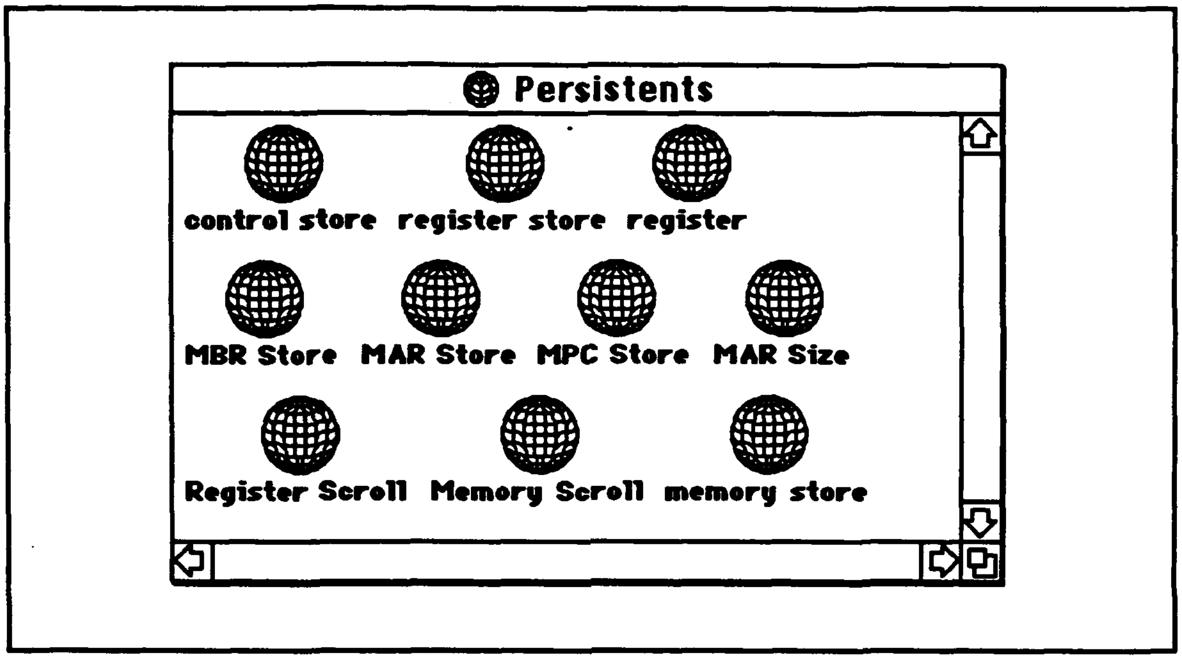


Figure 3.9 Cycle Universal Method

Program state in the Micro Simulator is maintained by using Prograph persistents. These persistents are presented in Figure 3.10. These persistents are initially loaded during the execution of the initial universal method, and get updated during various stages of subcycle execution.



Persistents

control store

register store

register

MBR Store

MAR Store

MPC Store

MAR Size

Register Scroll

Memory Scroll

memory store

Figure 3.10 Micro Simulator's Persistents

C. IMPLEMENTATION OF A SIMPLE COMPUTER (ASC)

Tanenbaum's microarchitecture was easily implemented with few modifications using the 'generic' class design presented in section A.3.b of this chapter. The best way to fully test this class design was to implement an architecture of a significantly different design. This section introduces another architecture called "A Simple Computer" (ASC) which is presented by Shiva (Shiva, 1991, pp. 220-273). A brief description of the design and operation of the ASC and the implementation of its simulator using the 'generic' classes refined in section B (revised description of 'generic' classes are located in Appendix A) is now presented.

1 Operation of the ASC Microarchitecture

A simplified block diagram of the ASC is presented in Figure 3.11. Like Tanenbaum's microarchitecture, the ASC has a datapath side and a control path side. The datapath side consists of three buses, various registers, memory bank (including

MAR/MBR), index register bank, constant registers (1 and -1 hard wired) and alu. All components on the datapath side of the ASC are sixteen bits wide, and numbers are represented using two's complement arithmetic. BUS1 and BUS2 direct data from the various registers into the ALU. BUS3 directs data from the output of the ALU back to the various registers.

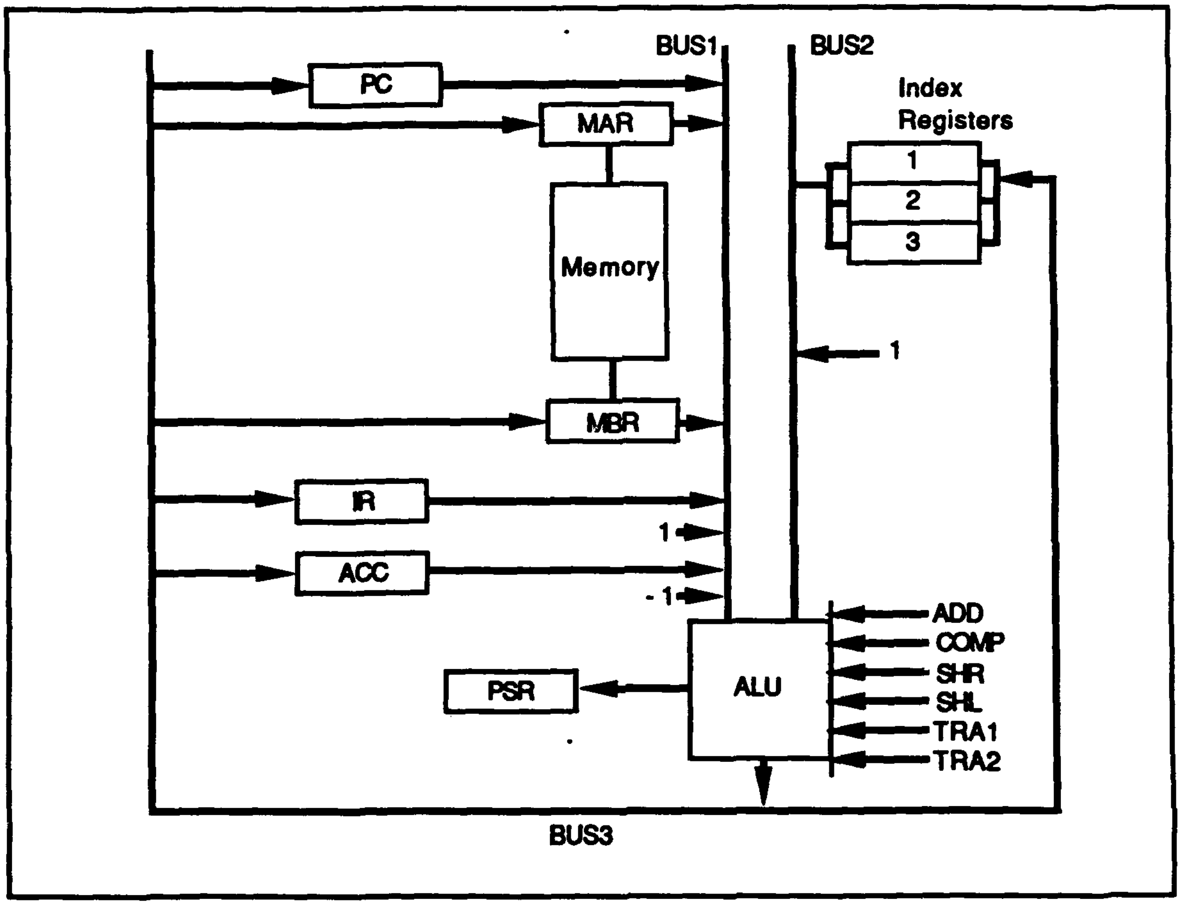


Figure 3.11 ASC Microarchitecture Block Diagram (Shiva, 1991, p.229)

Each register (other than the Index Registers) is a unique single entity. This means that all the appropriate control signals are routed to each of these devices separately (write to BUS1/2, read from BUS3). It should also be noted that not all registers can write to both BUS1 and BUS2. The design of the microprogram ensures that only one register at a time writes to each bus. BUS1 and BUS2 also have the sixteen bit constant values '1', while BUS1 also has the sixteen bit constant value '-1'. These 'values' act like registers with read only capability.

The ALU receives six signals from the microcontrol unit and receives sixteen bit data from BUS1 and BUS2. Only one control signal can be applied at a time. These control signals control the following functionality to the ALU:

1) **ADD**, add the values on BUS1 and BUS2 and place the results on BUS3.

2) **COMP**, take the two's complement of BUS1 and output the results to BUS3.

3) **SHR**, shift the value of BUS1 one bit to the right, with the high order bit replacing the low order bit, and output the results to BUS3.

4) **SHL**, shift the value of bus1 one bit to the left, replacing the low order bit with zero, and output the results to BUS3.

5) **TRA1**, directs the value of BUS1 to BUS3.

6) **TRA2**, directs the value of BUS2 to BUS3.

Notice that in the ASC design the shifter functionality is included within the ALU. The ALU also updates the value of the PSR (Processor Status Register). The PSR consists of 4 bits, C, N, Z, and O; these values stand for carry, negative, zero, and overflow respectfully. The ALU updates the PSR only when the accumulator register is written to. The overflow bit is set when the sum operation results in a number larger than 2^{15} -1. The negative bit is set when the result of an ALU operation is negative. The zero bit is set when the result of an ALU operation is zero. The carry bit is set when a carry out from the high order bit results from an addition operation.

The MAR and MBR registers are used as an interface between the memory and the CPU. When the MBR receives a READ signal it reads the contents of the memory pointed to by the MAR and place that location's value into the MBR. When the MBR receives a WRITE signal it will place the contents of the MBR in the location of memory pointed to by the value of the MAR. In this design it takes two complete machine cycles to read from or write to memory (i.e., register access is twice as fast as memory access).

The ASC design supports four types of addressing; direct, indexed, indirect, and indexed-indirect addressing (preindexed-indirect). The addressing modes are directly controlled by fields of the ASC's macroinstruction. The ASC's microarchitecture design relies specifically on this macroinstruction format. As mentioned above, all instructions used by ASC are sixteen bits wide. The ASC's instruction format is divided up into various fields as presented in Figure 3.12.

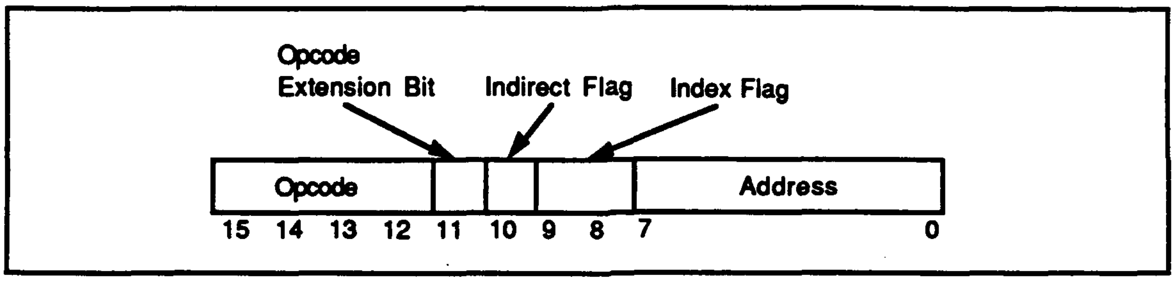


Figure 3.12 ASC Instruction Format

This implementation of the ASC supports 16 macroinstructions, thus four bits in the macroinstruction is needed to describe the opcode (bits 12-15). Bit 11, the opcode extension bit, can be used to increase the number of opcodes to 32. Bit 10, the indirect flag, is set when indirect addressing is used. The two bit index flag (bits 8 & 9) has two purposes: when the flag is set to '00 ', the index flag indicates that indexed addressing mode is not in use; when the flag is set to the values '01' through '11', it indicates that indexed addressing is in use with the corresponding index register. The eight bit address field (bits 0-7) is used for direct addressing, or combined with the other addressing modes. For a complete description of the actual instruction set refer to (Shiva, 1991, p.193).

The control side of the ASC microarchitecture consists of a Microcontrol unit, MPC, MIR, Decoder, and a Control Store. This configuration is presented in Figure 3.13.

The MPC points to the next line of the microprogram to fetch (from the control store). A microinstruction is 21 bits wide and can be in one of two different formats. The formats are distinguished by the high order bit. If the high order bit is zero, it is a type zero microinstruction. If the high order bit is 1 the instruction is a type one instruction. A type zero instruction actively sends control signals to the various components of the microarchitecture; each bit represents a control signal. A type one microinstruction uses input status signals to alter program flow of the microprogram. Therefore, a type one microinstruction will cause the MPC to be set to some address other than the next instruction in the control store. For a more detailed description of the microinstruction formats and control signal outputs see (Shiva, 1991, pp. 267-268).

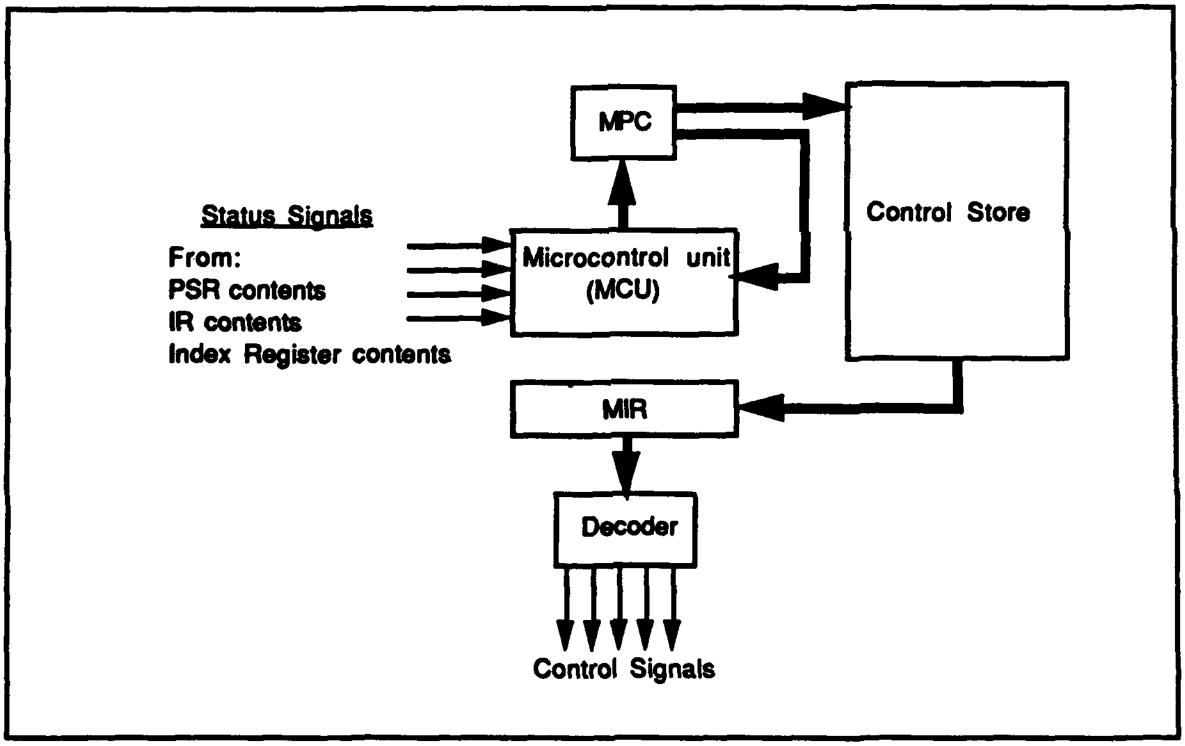


Figure 3.13 Block Diagram of ASC Microcontrol Hardware

The Microcontrol unit receives status signals from the PSR, instruction register (IR) and index registers. Based on the status signals and the type of the microinstruction, the microcontrol unit will either load a new address into the MPC and load a new microinstruction into the MIR, or it will take the present contents of the MIR and decode it into control signals, increment the MPC, and repeat the process.

2. Design of The Class Hierarchy

The design of a class hierarchy to implement a simulation program for the ASC microarchitecture also begins with the 'generic' microarchitecture classes presented in Appendix A and building from them into a full Macintosh application. Appendix D contains the Prograph source code listing for the simulation of the ASC microarchitecture.

a. Review and Modification of the General Class Hierarchy

This section reviews each class defined in Appendix A ('generic' classes) and describes the modifications necessary to implement these classes in the ASC design.

No modifications (from Appendix A) were required to the following classes: **CONTROL STORE, MAR, MBR, MPC, MEMORY LOCATION, MEMORY BANK, REGISTER, REGISTER BANK, SHIFTER, STORAGE BANK,** and **STORAGE LOCATION.** The classes **MUX** and **MICROINSTRUCTION** were not used. **MUX** was not used because the ASC design contains no mux's. The **MICROINSTRUCTION** class was not used because simple instances of register were used to hold microinstructions. The **ALU** class includes message passing to the shifter class in its **math** method. This enables shifter functionality in the ALU in accordance with the ASC microarchitecture design. The following classes were modified or added:

Class: ALU (modified)
modification:
1) Added the method **overflow** to determine if the output of the ALU is causing an overflow condition.
2) Modified the method **math** to account for the ASC specific functionality (actual operations) including the updating of the **PSR.**

Class: MICRO CONTROL (added)
Superclass: none
Variables: none
Methods: mcu, read control store, Bus1 Data,
Bus2 Data, Bus3 Signals,
ALU Signals, Memory Signals.
Description: Contains methods to read microinstructions, generate control signals, and branch microprogram control flow.

Class: MIR (modified)
modification: Added the methods decode 0, and decode 1 to decode the type zero and one microinstructions.

Class: INDEX BANK (added)
Superclass: STORAGE BANK
Variables: none
Methods: zero index
Description: Describes the data structure and methods necessary for implementing an index bank for the ASC. The zero index method generates a signal to determine if the contents of an index register is zero.

Class: IR (added)
Superclass: REGISTER
Variables: none
Methods: parse
Description: The parse method separates the contents of the instruction register into the various fields.

Class: PSR (added)
Superclass: REGISTER
Variables: none
Methods: decode
Description: Contains an instance variable that maintains the value of a status register. Includes a method to decode its contents for use in the microcontrol unit

The source code listing in Appendix D also includes the various system classes supplied by prograph and the class **sim**, as discussed in Section B.2.a.

3. Design of the ASC Simulator

a. The user interface

The ASC simulator, like the Tanenbaum simulator, was implemented using Prograph on the Macintosh microcomputer. This section discusses the general design of the ASC simulator and its user interface.

Figure 3.14 presents the menu bar for the ASC simulator. The ASC simulator application menu bar and most of its controls have the same functionality as the Tanenbaum micro simulator menu bar. Since the ASC does not use a general purpose register bank, the **Alter Register** menu selection under the **Controls** menu was removed. This menu selection allows the user to enter a value from the keyboard by entering the register number and its new value. The **Register** menu was added to the menu bar to enable keyboard entry changes for the program counter (**Update PC**).

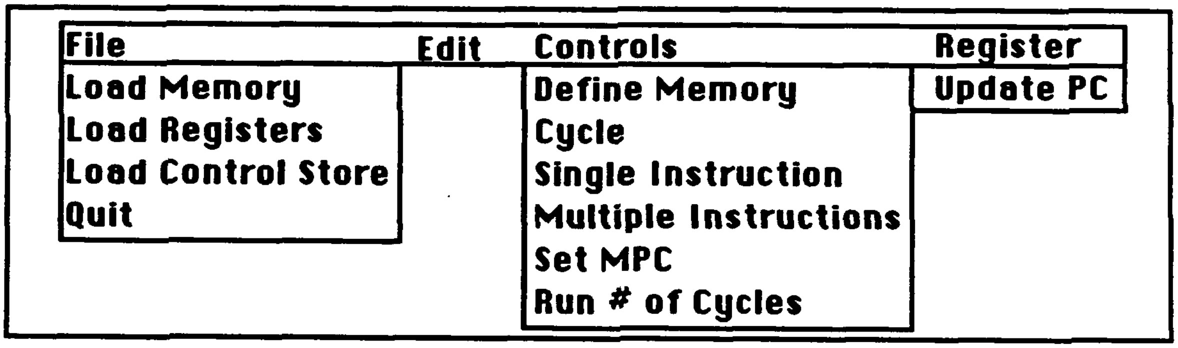
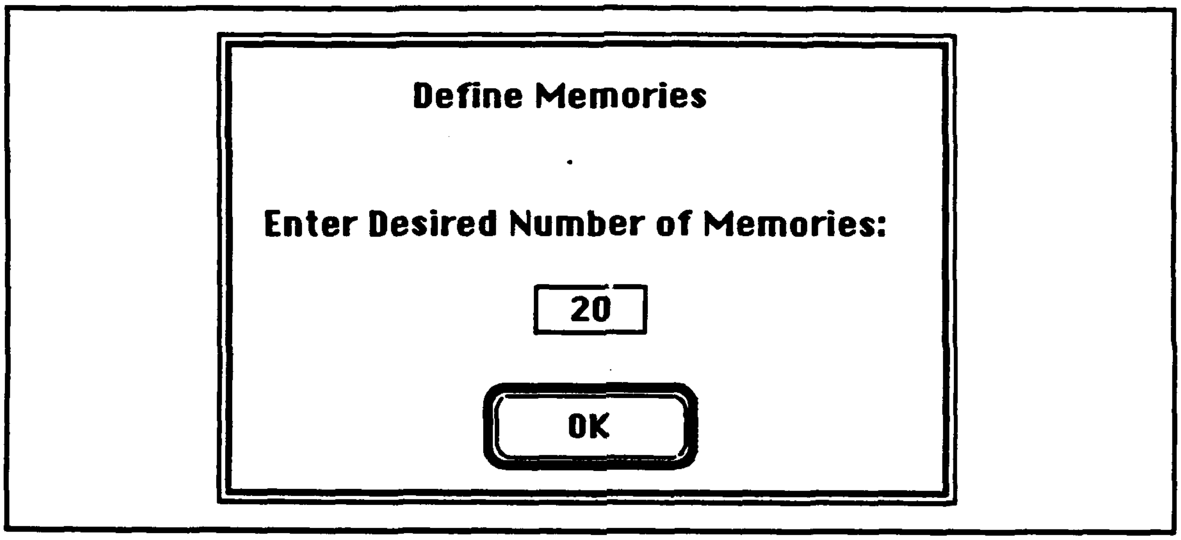


Figure 3.14 ASC Simulator's Menu Bar

The **File** menu is exactly the same as the Tanenbaum Micro Simulator, and its selections provide the same functionality. The required data format for the input text files is also the same. The **Edit** menu selection contains the standard Macintosh editing features.

The **Define Memory** function was changed because the ASC simulator was designed to operate with a fixed memory/register bus width of 16 bits. Also, since the

ASC uses individual registers, there is no need to specify the required number of registers in the register bank. This selection still initializes the memory bank to the desired number of memories. This resulted in a **Define Memory Dialog Box** with one input, 'Enter Desired Number of Memories:'. This dialog box is presented in Figure 3.15. All other menu selections have the same functionality as the Tanenbaum micro simulator.

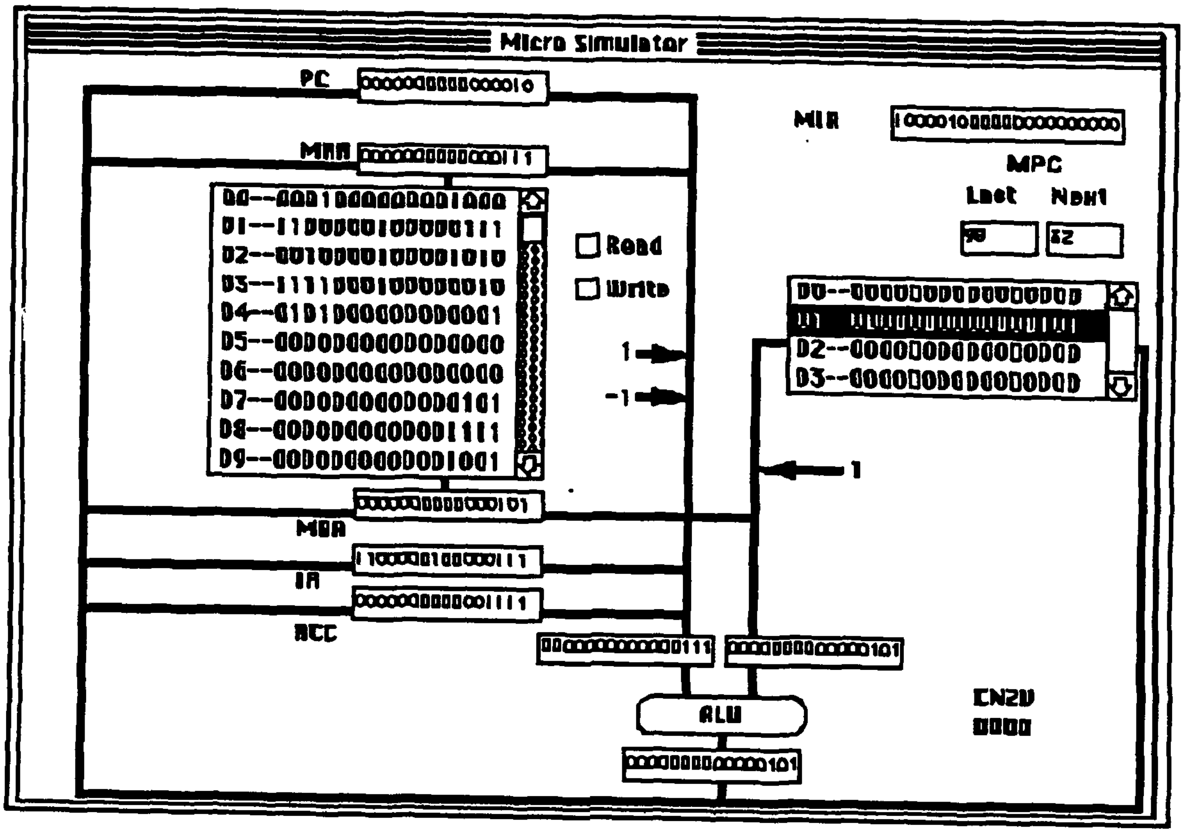


Define Memories

Enter Desired Number of Memories: 20

Figure 3.15 The Define Memory Dialog Box (ASC)

Status of the simulator is displayed in a Window (Micro Simulator) which includes a diagram of the data side of the microarchitecture along with the values of the various components. A reduced copy of this window is presented in Figure 3.16. The values in the various boxes indicate the value of the respective components after each microinstruction is executed. Several new items were added: MIR, Index Bank, CNZV (PSR), and the constant values (1 and -1). The CNZV output gives the status of the PSR, while the constant values are placed on the associated input buses by the microcontrol unit. The MIR box gives the bit string of the last microinstruction executed.



```
Micro Simulator
PC   0000000000000010
MAR  0000000000000111
D0--0001000000001000
D1--1100000100000111
D2--0010000100001010
D3--1111000100000010
D4--0101000000000001
D5--0000000000000000
D6--0000000000000000
D7--0000000000000101
D8--0000000000001111
D9--0000000000001001
MBR  0000000000000101
IR   1100000100000111
ACC  0000000000001111
MIR  10000100000000000000
MPC  Last 58   Next 52
D0--0000000000000000
D1  0000000000000000
D2--0000000000000000
D3--0000000000000000
0000000000000111   0000000000000101
ALU
0000000000000101
CN2V
0000
```

Figure 3.16 The ASC Simulator Window

b. ASC Simulator Program Structure

The ASC simulator's main program uses the universal method Cycle (as in the Tanenbaum simulator), but the ASC's design does not use subcycles. The Cycle universal method is the main program and brings together all the various classes' methods to work as a complete simulator. The other universal methods used in the Tanenbaum simulator were also integrated seamlessly in the ASC simulator (generic conversions between bit strings and boolean strings, etc). Program state is maintained by using Prograph persistents (see Appendix D).

D. COMPARISON OF THE TWO SIMULATORS

Both architectures possess many similarities. Their designs have a classic layout that consists of a data path and a control path. The data path side consists of some type of register configuration (including a program counter, instruction register, and accumulator), memory with a MBR/MAR interface, buses, and a two bus input ALU. The data path side for these architectures is 16 bits wide. On the control side, both architectures have a control store, MPC, and MIR. Both architectures use a microprogram to control their various components for the execution of macroinstructions. Since they both use a microprogram, their macro level instruction set can be expanded or changed by altering their respective microprograms. With respect to the implementation of the simulators, both designs use the same format for textfile input of the macroprogram to the memory. Both simulators allow altering the microprogram by loading the control store with a textfile representing the microprogram.

Although the Tanenbaum and ASC designs are very similiar, there are also some differences in their designs. One major difference between the Tanenbaum design and the ASC design is the layout of the registers. With the ASC, each register (other than the Index Registers) is a unique single entity. This means that all the appropriate control signals are routed to each of these devices separately (write to BUS1/2, read from BUS3). The Tanenbaum design uses a bank of registers which allows two registers at a time to send their values to the A and B Buses and one register to receive the contents of the C Bus. The registers are selected by sending the appropriate register numbers to the register bank. Thus, the ASC design requires many more control signals to manipulate the registers. In the ASC design not all registers can write to **both** BUS1 and BUS2, while the Tanenbaum design allows the same operations for all registers. The design of the microprogram ensures that only one register at a time writes to each bus. The implementation of ASC requires individually initializing each of the separate registers, where the Tanenbaum simulator simply inializes the entire register bank with one operation. In the ASC design, BUS1 and BUS2 also have the sixteen bit constant values '1', while bus 1 also has the sixteen bit constant value.'-1'. These 'values' act like registers with read only capability. The ASC design uses an index register bank to support indexed addressing. The functionality of the index bank is similar to the Tanenbaum register bank. This required the addition of the **index bank** class in the ASC simulator. This class was added to support the additional functions of indexed addressing.

While both designs support direct addressing, other addressing modes are not shared by the two microarchitectures. The Tanenbaum design supports immediate addressing and includes a stack, along with stack operations such as push and pop. Immediate addressing is supported completely through the microprogram. The implementation of the stack does not require any special simulator features except a register reserved for a stack pointer (which is a general register in the Tanenbaum design) and the appropriate microprogram support.

The ASC design supports direct, indirect, indexed addressing, and indexed-indirect adressing. Direct addressing is similiar to the Tanenbaum design. Indirect, and indexed addressing is supported by adding the method **parse** to the new class **IR** (Instruction Register). The **parse** method decodes special fields that direct the Microcontrol unit to use indirect, direct, or indexed-indirect addressing.

The microinstruction format of the two designs differs of course, but how the formats are used is also different. In the Tanenbaum design, all microinstructions are of the same type; a microinstruction is decoded and the various control signals are sent to all the components. Part of the microinstruction field contains a conditional that, based on the microsequencer output, will cause the Mmux to branch the microprogram or go to the next microinstruction. In the ASC design, there are two types of microinstructions: a type zero microinstruction, which sends control signals to the various components; and a type one microinstruction, which controls branching of the microprogram. This difference in the microinstruction format required various additions/alterations to the 'generic' classes. The **MIR** class was modified to support the decoding of the two types of microinstructions. The class **MICRO CONTROL** was added (in lieu of the **MICRO SEQUENCER** class which was removed from the Tannenbaum simulator) to support the microcontroler functionality specific to the ASC design. This type of change would be required when switching between any different architectures. In the ASC design the Micro Control unit makes all deciscions involving branching; this eliminates the need for a Mmux as found in the Tanenbaum design.

The execution of macroinstructions also differs between the two designs. The Tanenbaum design loads the macroinstruction into an instruction register. As the microprogram progresses, the macroinstruction is shifted to the left. The microprogram branches to different locations based on the value of the most significant bit of the macroinstruction. This means that, depending on the size of the opcode of the macroinstruction, the microprogram can branch for up to 8 cycles to parse the macroinstruction (the largest opcode is 8 bits). The decoding of an ASC macroinstruction is a much shorter process. The macroinstruction is fetched, and placed in the instruction register. The opcode of the macroinstruction matches the address of the microcode required to execute the macroinstruction. This results in a slightly larger microprogam, but fewer cycles are needed for each macroinstruction. These differences are accounted for in the respective simulators by microcode and their respective objects (**MICRO SEQUENCER** in Tanenbaum, and **MICRO CONTROLLER** in ASC).

The Tanenbaum design sends three signals to the micro sequencer: N (ALU output is negative); Z (ALU output is zero); and COND (microprogram branching conditions based on N and Z). The ASC design is slightly more complicated. It maintains a PSR (Processor Status Register) that has several bits (C, N, Z, and O as discussed in section C.1) representing the status of the number in the accumulator. There are more inputs to the micro controller, PSR, IR, and Index Registers contents. These differences are supported in the respective classes **MICRO SEQUENCER** and **MICRO CONTROLLER.** Also, the classes, **IR,** and **PSR** with associated methods were added to the class hierarchy for the ASC simulator.

The two designs have different ALU functionality in that they have different inputs and operations. The Tanenbaum design has two different components, an ALU and a Shifter. The ASC combines the functionality of these two components into the ALU. The method **overflow** was added to the class **ALU** and the **math** method was altered to provide the required functionality for the ASC design. In the ASC design, messages (from within the ALU methods) are sent to the shifter object to give the effect of the shifter residing inside the ALU.

Both designs have several small variations in some of the objects as discussed above. The various objects are brought together to form a complete simulator via the main program. Since these simulators differ, the main programs differ. The main programs for both simulators are called 'cycle' and are implemented as universal methods (in Prograph). The user interfaces differ in appearance (because the buses and components differ), but the approach for the construction of the user interfaces are exactly the same. Their menu bars and dialog boxes are almost the same; this allowed a great deal of code to be reused. These two simulators were designed by first considering the Tanenbaum design, making 'generic' classes, and then producing the Tanenbaum simulator. The ASC simulator was designed by taking the 'generic' classes previously derived and applying the changes necessary to give ASC functionality. This process could have easily been reversed with very little inpact on the final result.

E. SUMMARY

A 'generic' class hierarchy was designed for the application of simulating a general purpose computer microarchitecture. A simple computer microarchitecture (Tanenbam's) was introduced in which a simulator was designed and built using this 'generic' class hierarchy. Few modifications/additions were required in building the Tanenbaum simulator from the 'generic' classes. To further test the 'generic' class design, a second computer microarchitecture was introduced (Shiva's) in which a simulator was designed and built using the refined 'generic' class hierarchy arrived at when designing the Tanenbaum simulator. Once again it was discovered that few modifications were required in the refined class hierarchy when extending its use in another simulator.

IV. SUMMARY, CONCLUSIONS AND RECOMMENDATIONS FOR FUTURE RESEARCH

A. SUMMARY

Chapter I developed the need for simulation of computer architectures at various levels. It introduced the notion that architecture simulation could be more easily implemented using object-oriented design and programming. The chapter further developed object-oriented concepts by giving basic definitions and terminology. Basic microarchitecture components and operation were then described using an example microarchitecture. Finally, Prograph, an object-oriented, visual, data-flow language was introduced, along with a basic description of its syntax and use applied to object-oriented programming.

Research areas related to this thesis were discussed in Chapter II. These areas included: architecture simulation for educational purposes, class hierarchy design for simulation of multimicrocomputers, object-oriented approach to VLSI routing, object-oriented approach to interactive user interface for microprogram simulators, and work being done using object-based languages such as Ada. Several of these papers pointed out that the design of the class hierarchy is a crucial portion of the design of any simulator. There is a definite interest in the areas of classroom instructional simulators using object-oriented programming languages with reusable software components and an easy to use user interface. As of yet, however, there is no work encapsulating all of these concepts simultaneously. This provided the motivation for this research.

Chapter III showed how an existing object-oriented design methodology was used in designing a 'generic' class hierarchy to implement the components of the basic computer microarchitecture introduced in Chapter I. Tanenbaum's microarchitecture design and operation was then introduced. The 'generic' classes were used in the development of a microarchitecture simulator using Tanenbaum's microarchitecture. It was found that very little modification to the existing 'generic' class hierarchy was required in implementing the Tanenbaum simulator. The design and operation of Shiva's ASC microarchitecture was also introduced. A simulator for this microarchitecture was implemented to further test the usefulness of the 'generic' microarchitecture class hierarchy. Once again, only a few modifications to the 'generic' class hierarchy were required to implement this simulator. The design and implementation of the two simulators, including the user interfaces, were also discussed.

B. CONCLUSIONS

Object-oriented programming provides a natural environment for modeling and simulation problems. This is because the implementation details are hidden, allowing objects to reflect the real-world environment. A careful class hierarchy design is, however, essential to the development of any object-oriented program.

'Generic' classes can be created to simulate the various objects found as components in most computer microarchitectures. Careful design of these component objects allows the reuse of the code for many different simulators. This was demonstrated through the development of simulators for two different microarchitectures. In implementing the two simulators, it was still necessary to write a 'main' program that puts the various 'generic' objects together to form a complete simulator.

It was also found that components like ALU's, which are peculiar to each architecture, require complete remodeling; there was very little code reusability. Parts of an ALU model can be reused, like adders, but the control signals are normally different enough to warrant a complete redesign.

Each simulator required slightly different user interfaces; however, these user interfaces had almost identical menus and functionality. They only had a different depiction of the buswork and components.

The microarchitecture simulators were implemented in Prograph, a visual, object-oriented, data flow language operating on the Macintosh. It was found that although there was an initial learning curve (to adapt to the unaccustomed nature of the pictorial syntax), Prograph made the implementation of the class hierarchy for the simulators relatively easy. Prograph's application builder (used to generate user interfaces) allowed rapid development of the user interfaces. Another advantage to Prograph is that it is relatively inexpensive and readily available.

Shiva's ASC microarchitecture simulator was demonstrated to an introductory computer organization class studying the ASC microarchitecture. The class found the simulator to be quite helpful in learning the concepts of the architecture as they could trace through complex microinstructions in a short period of time without having to keep track of all of the parameters.

C. RECOMMENDATIONS FOR FUTURE RESEARCH

There are several areas of research that logically follow from this work. As is often the case in research, more questions were raised than answered. With a basic 'generic' class hierarchy designed, it is possible to pursue experimenting with 'families' of architectures. The object-oriented approach should prove to be ideal for this too in that one should be able to implement the 'lowest' member of a family (such as the 68000 microprocessor in the series of 680x0 microprocessors) and inherit the features of that architecture as one moves to other more advanced members of the same family.

Even though this research was performed using Prograph, which was found to be an excellent language for the development of these systems, it should also be very interesting to investigate other object-oriented (or object-based) languages for implementing microarchitecture simulators and comparing the development effort with the results of this thesis.

The simulators in this thesis used text files representing the object code of the macroprograms and microprograms. The micro/macroprograms were assembled by hand and represented in the text file using ones and zeros. The usefulness of the simulators can be increased by developing micro/macro assemblers which would generate text files compatible with the simulators developed in this thesis. It should be possible to use object-oriented techniques to develop 'generic' classes which model various assemblers for different simulators.

Although these simulators were very useful for classroom instruction, simulators are most often used in designing actual systems. For a simulator to be useful, it is necessary to build into the components some automatic monitoring functions. An example would be a counter that determines how many times memory is accessed for each macro level instruction execution. One could also include in each object a facility to maintain an 'execution history' that would record the usage of components and the frequency of each component's use.

As previously mentioned, a new ALU class had to be designed and coded for each microarchitecture implementation. Research into designing a 'generic' class hierarchy in support of ALU design would be another area of challenging research. This would require the specification of control signal inputs, functions based on control signals, and status outputs. All vary greatly among different ALU's.

With growing interest in multiprocessors, it would be logical to persue multiprocessor simulators. This would require investigating timing effects of all operations involving each component. After obtaining timing information, new data structures will have to be introduced to account for timing constraints.

Finally, the development cycle when using the simulator to design a microprogram could be shortened by integrating a microprogram editor with the simulator. Presently the programmer is required to write the microprogram, load it into the simulator, and then run the simulator. The user then has to evaluate any required changes, stop the simulator, edit the microprogram text file and repeat the process. This could be simplified if the simulator were integrated with a microprogram editor.

APPENDIX A

This Appendix contains the revised 'generic' classes derived when implementing the Tanenbaum design microarchitecture simulator.

Class: ALU
Superclass: none
Variables: none
Methods: and, or, not, math, zero?, positive?
Description: Represents a combinational circuit; thus, it has no state. Methods are provided that perform logical operations on a stream of input data.

Class: CONTROL STORE
Superclass:
Variables: contents:
(array of MICROINSTRUCTIONs)
Methods: load
Description: Holds the entire microprogram. It must be loaded with the microprogram prior to any simulation execution.

Class: MAR
Superclass: REGISTER
Variables: none
Methods: mar
Description: Contains an address of a **MEMORY LOCATION** within a **MEMORY BANK.** The method **mar** accepts a control signal which determines if a value is to be stored into the MAR.

Class: MBR
Superclass: REGISTER
Variables: none
Methods: mbr
Description: Models the data interface to the memory bank. The method **mbr** accepts a control signal which determines if the data input will be written to the MBR.

Class: MEMORY BANK
Superclass: STORAGE LOCATION
Variables: contents: array of MEMORY LOCATIONs
Methods: load, read, write
Description: A read or write to a **MEMORY BANK** implies a **read** or **write** to a specific **MEMORY LOCATION** contained in the **MEMORY BANK.** The

method load is used to load a user program or data into the MEMORY BANK.

Class: MEMORY LOCATION
Superclass: STORAGE LOCATION
Variables: none
Methods: none
Description: Used to describe the contents of a location in a MEMORY BANK.

Class: MICROINSTRUCTION
Superclass: none
Variables: instruction (string of bits)
Methods: none
Description: Defines data structure that describes a microinstruction.

Class: MICRO SEQUENCER
Superclass: none
Variables: none
Methods: generate signal
Description: Simulates the micro sequencer component of Tanenbaum's architecture. The method **generate signal** sends a signal to the mux for controlling logic and addressing of the MPC.

Class: MIR
Superclass: REGISTER
Variables: none
Methods: decode
Description: Contains the various control signals. The method **decode** is used to parse the register contents into required control fields.

Class: MPC
Superclass: REGISTER
Variables: counter, cycles
Methods: set, increment, jump, set cycles,
get counter
Description: Models a microprogram counter.

Class: MUX
Superclass:
Variables: none
Methods: mux
Description: Models a combinational circuit. The output is one selection from many inputs.

Class: REGISTER
Superclass: STORAGE LOCATION
Variables: . none

Methods: none
Description: Defines how the data is represented in a system, (i.e., how many bits represents a register/memory location).

Class: REGISTER BANK
Superclass: STORAGE BANK
Variables: contents: array of REGISTERs
Methods: load
Description: Similar to a **MEMORY BANK.**

Class: SHIFTER
Superclass: none
Variables: none
Methods: shift left, shift right, no shift
Description: Models a combinational circuit. Methods take a binary input and perform a binary shift to the left or right.

Class: STORAGE BANK
Superclass: none
Variables: contents:
array of STORAGE LOCATIONs
Methods: initialize, load, read, write, binary read, binary write
Description: Defines data structure and methods for modeling a general storage object. Allows for access using both integer and binary address references.

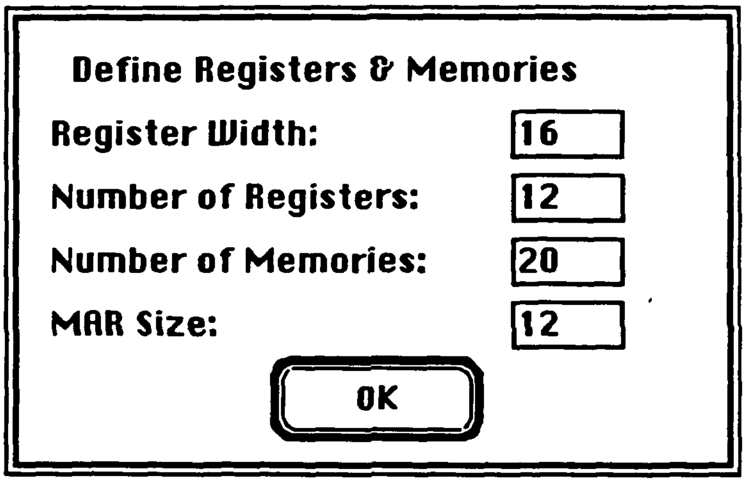
Class: STORAGE LOCATION
Superclass: none
Variables: contents: string of bits
Methods: initialize, read, write
Description: Provides methods for accessing (read/write) a storage location in a memory or register bank.

APPENDIX B

This appendix contains a sample user session for the Micro Simulator and the ASC Simulator.

Tanenbaum Micro Simulator:

The Micro Simulator is started by double clicking on the application icon labeled **Micro Simulator**. This causes the application to load, the initialization of the menu bar, and the following dialog box to appear:



Define Registers & Memories

Register Width: 16

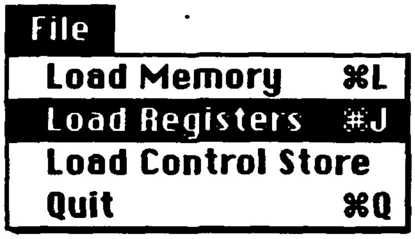
Number of Registers: 12

Number of Memories: 20

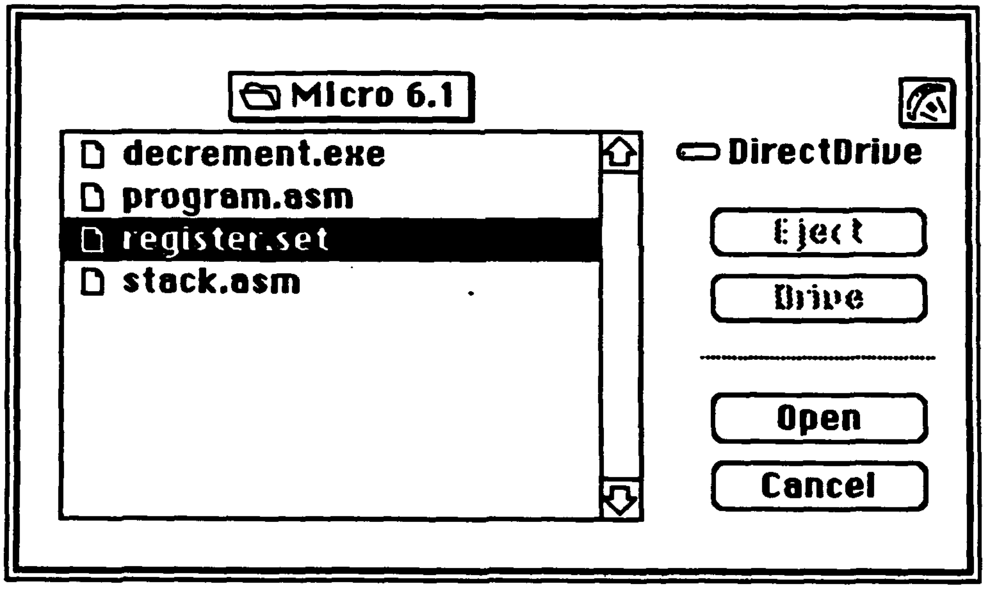
MAR Size: 12

The user is required to enter values for each field in the dialog box (the above values are defaults). After all values have been entered, the user presses the Enter key or clicks the OK button. In this example, the Register Width field configures the simulator to have a 16 bit data path for buses, registers, and memory. The Number of Registers and Number of Memories fields configure the simulator to have the respective values simulated. In this case, 12 registers and 20 memories have been selected. The MAR Size field specifies the number of bits the MAR will use for addressing the memory. These bits are the low order bits passed from the data path to the MAR; this example specifies a 12 bit MAR.

The Tanenbaum design uses a set of registers, including general purpose, special purpose, and constant values (registers 6, 7, 8, and 9). Register values can either be loaded into their respective registers individually by using the Alter Register command (discussed below), or they can be loaded from a text file by using the **Load Registers** command in the **File** menu. To initialize the registers using a text file, select the **File** menu (click or command-J), its choices are shown below:



Choose the **Load Registers** selection (this can be done by either clicking with the mouse or using the command-J key sequence). This action displays the file selection dialog box as shown below:

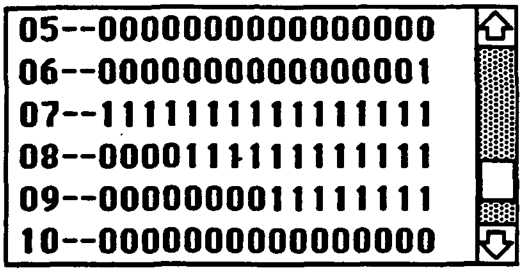


Select the name of the text file that represents the desired register configuration (in this case **register.set**). There are no restrictions on naming conventions for any text files associated with this simulator. The two buttons **Eject** and **Drive** are disenabled (cannot be used) because there were no other drives mounted during this example. The contents of the **register.set** text file is shown below:

```
0-0000000000000000    Program Counter
1-0000000000000000    Accumulator
```

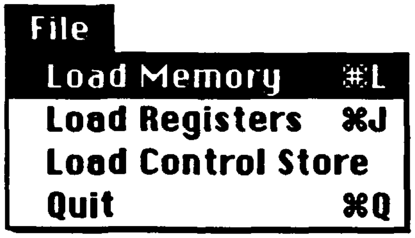
```
2-0000000000000000    Stack Pointer
3-0000000000000000    Instruction Register
4-0000000000000000    Temporary Instruction Register
5-0000000000000000    0
6-0000000000000001    +1
7-1111111111111111    -1
8-0000111111111111    AMASK   0FFF    (hex)
9-0000000011111111    SMASK   00FF    (hex)
```

The text file format for both register and memory descriptions is the same: any number (representing the location or address), followed by a dash ('-'), followed by 1's and 0's (which represent the bit string), followed by a comment The numbers in the text file are for the use of the programmer only; the simulator loads all binary strings in order starting at location zero. The results of loading this text file into the simulator are shown in the register scroll box below (found in the simulator window, this example shows locations 5-10).

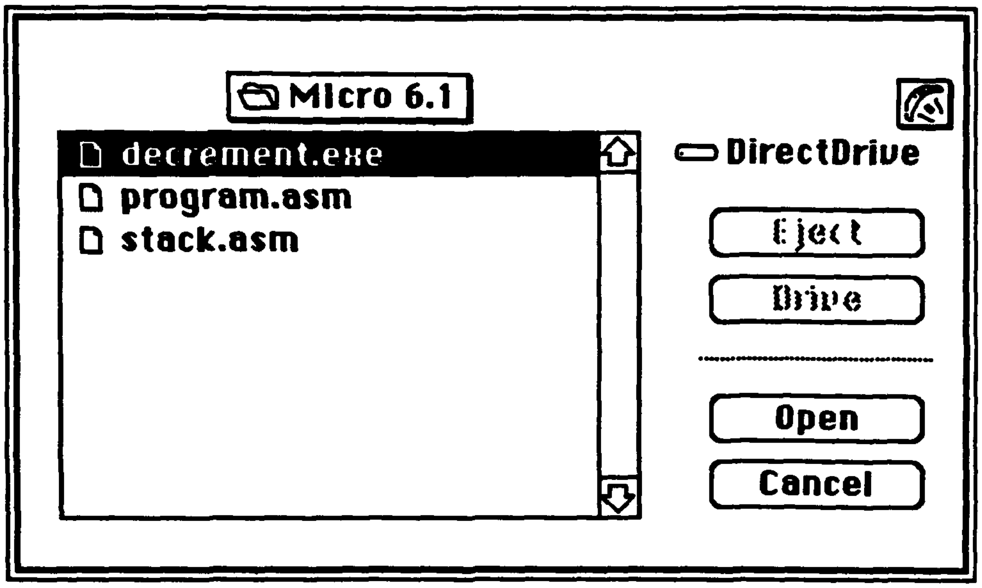


```
05--0000000000000000
06--0000000000000001
07--1111111111111111
08--0000111111111111
09--0000000011111111
10--0000000000000000
```

Next, load an object code program into the simulator's memory (this is a text file previously saved). This is done by choosing the **Load Memory** selection from the **File** menu as shown below (by clicking or command-L key combination):



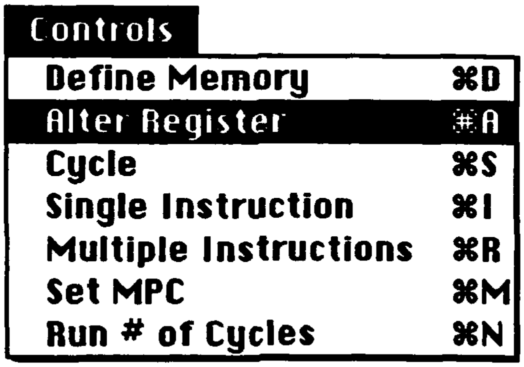
This causes the file input dialog to be displayed. Selection of the **decrement.exe** file is shown below:



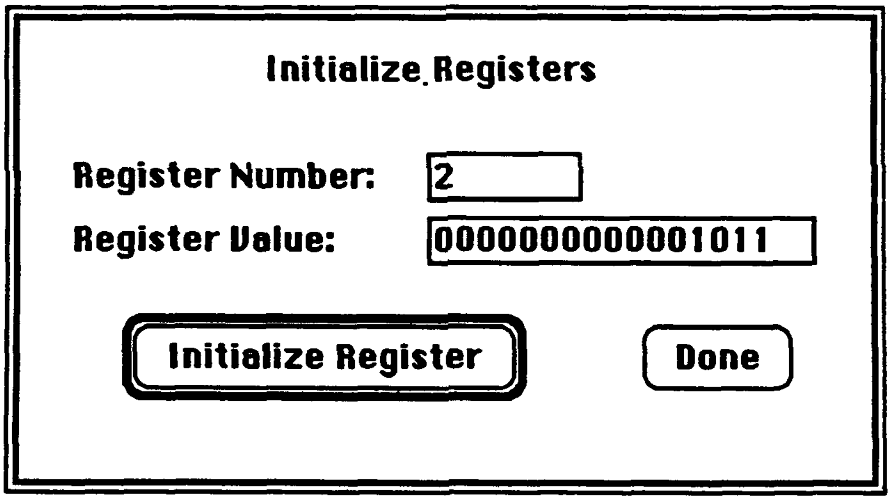
The **decrement.exe** program is a simple program that loads a constant '4' into the accumulator, pushes it onto the stack, decrements the accumulator value, and continues four more times. After completing this task the program remains idle by reaching step '5' and then continuously branching back to step '5'. The text file of this program is shown below:

```
00-0111000000000100  LOCO 4          Loads the constant 4
01-1111010000000000  PUSH            Push AC onto the stack
02-0011000000001100  SUBD MEM[12]    Subtract memory loc #12
03-0101000000000101  JZER 5          if AC = 0 then PC := 0
04-0110000000000001  JUMP 1          Set PC := 1
05-0110000000000101  JUMP 5          Stay running idle
06-0000000000000000  **
07-0000000000000000  **
08-0000000000000000  **
09-0000000000000000  **
10-0000000000000000  **
11-0000000000000000  **
12-0000000000000001                          Constant 1
```

Now that the program is loaded into the memory, it is necessary to determine where the stack pointer will point (recall that only 20 memory locations were defined in this example). In this example the initial stack pointer location will be initialized to '11'. This is done by using the **Alter Register** command under the **Controls** menu. This operation is shown below:



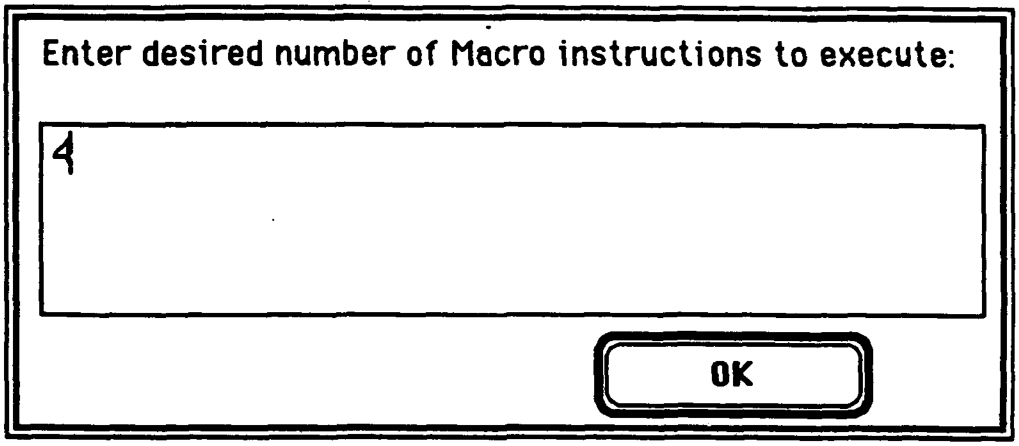
Selecting the **Alter Register** operation displays the dialog box below. We want to change the stack pointer (register 2) to a value of eleven. A 2 is entered in the **Register Number** field, and the appropriate string of 1's and 0's are entered into the **Register value** field. Press the **Initialize Register** button to enter the value into the register. The dialog box will remain displayed so that other entries can be made. Press **Done** to remove the dialog box.


Initialize.Registers

Register Number: 2

Register Value: 0000000000001011

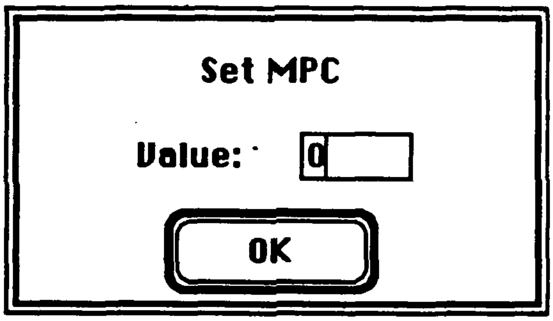
At this point the user program is ready is ready to execute. Execution can proceed by one of several methods which are selected from the **Controls** menu. The **Cycle** command causes the simulator to execute one microinstruction. The **Single Instruction** command causes the simulator to execute the necessary microinstructions to complete one macroinstruction. The **Multiple Instructions** command displays a dialog box requesting the desired number of macroinstructions. The user enters this value and the requested number of macroinstructions are executed. The dialog box for this operation is shown below:



Enter desired number of Macro instructions to execute:

4

The **Set MPC** command of the **Controls** menu is used to reset the MPC to an input value (to alter microprogram flow). The dialog box for this operation is shown below:

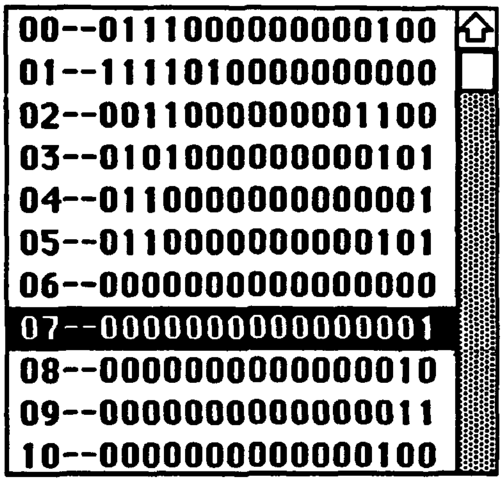


Set MPC

Value: 0

The **Run # of Cycles** command of the **Controls** menu displays a dialog box similar to the **Multiple Instructions** command, except the simulator executes the desired number of microinstructions.

The sample program can be run with any combination of the above commands. The diagram below shows the contents of memory after the **decrement.exe** program has run to completion:



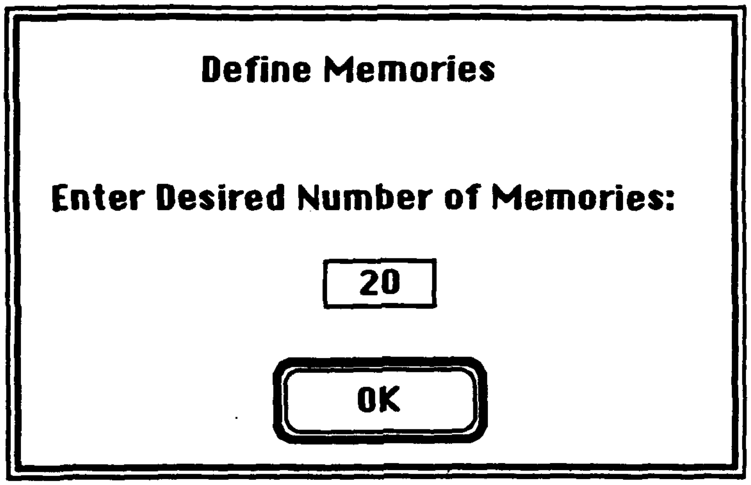
```
00--0111000000000100
01--1111010000000000
02--0011000000001100
03--0101000000000101
04--0110000000000001
05--0110000000000101
06--0000000000000000
07--0000000000000001
08--0000000000000010
09--0000000000000011
10--0000000000000100
```

Referring to the above figure, it shows that the numbers 4, 3, 2, and 1 were loaded into the memory locations 10, 9, 8, 7 respectively. Notice that the first location loaded by the stack pointer is 10 (recall that the stack pointer was initialized to 11). This is because the stack pointer is decremented before the value is written into memory. The above figure also shows that location 7 is highlighted; the simulator highlights the last value written to (in both the register and memory banks).

ASC Simulator:

The ASC simulator is very similar to the Tanenbaum simulator. Most of the menu operations have the same effects. The following is a sample session with the ASC simulator.

The simulator is started by double-clicking on the icon named ASC. This causes the application to load, the menu bar to initialize, and the following dialog box to appear:

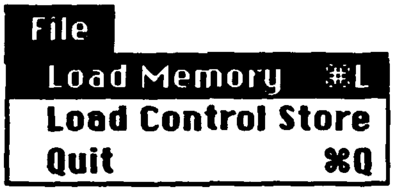


Define Memories

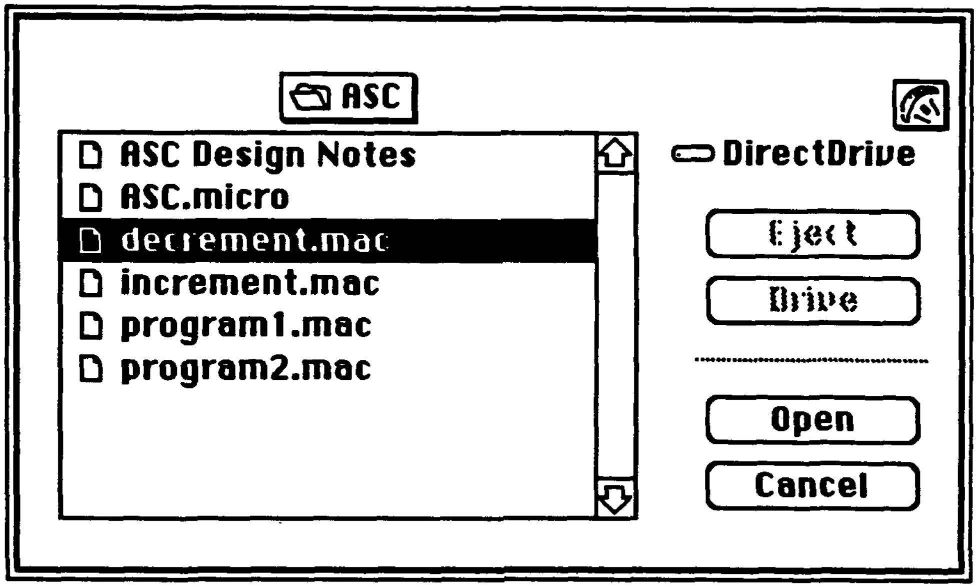
Enter Desired Number of Memories: 20

This dialog box simply initializes the number of memory locations, in this case, 20. In this simulator the bus, register, and memory widths are hard coded to 16 bits. All of the registers have specific purposes, as discussed in the thesis body.

The next step is to load the sample program into memory. This is done by selecting the **Load Memory** command from the **File** menu as shown below:



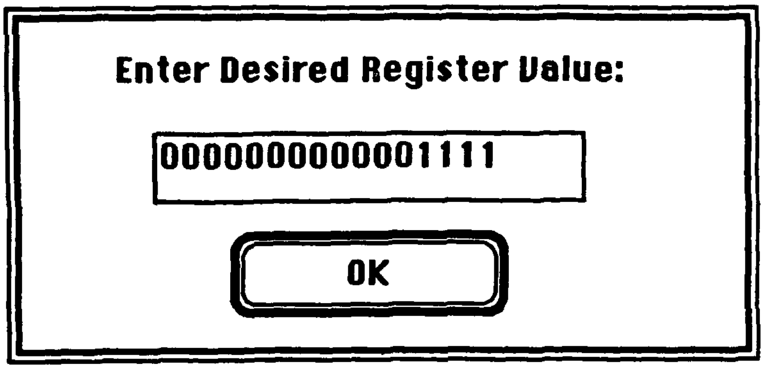
This action displays the file selection box shown below:



Selecting the **decrement.mac** text file as shown above (previously saved text file) loads the text file contents into the memory. The contents of **decrement.mac** are shown below. This program loads the accumulator with the value '15' (located at memory 8). It also loads index register #1 with the value '5'. It places the contents of the accumulator into the memory locations 11-15, then the program repeats. This simulator uses the same type of text file formats as the Tanenbaum simulator. Similar actions can be executed by choosing **Load Control Store** from the **File** menu. This will load a microprogram represented by the text file into the control store.

```
00-0001000000001000  LDA W/MEM[8]
01-1100000100000111  LDX INDEX 1 W/MEM[7]
02-0010000100001010  STA 10,1
03-1111000100000010  TDX INDEX 1 BRA 2
04-0101000000000001  BRU 1
05-0000000000000000  **
06-0000000000000000  **
07-0000000000000101  CONST 5
08-0000000000001111  CONST 15
09-0000000000001001  CONST 10
```

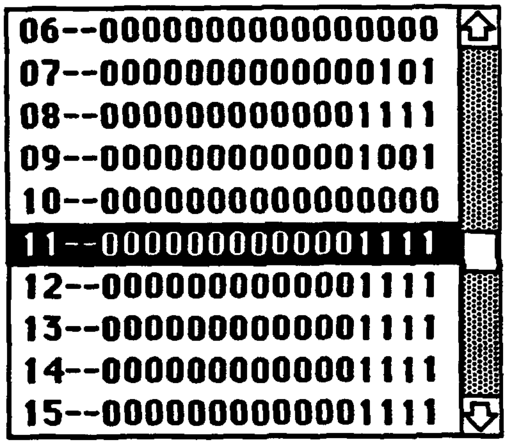
At this point the program is ready to execute. The ASC program execution controls operate with the same functionality as the Tanenbaum simulator. The **Controls** menu is the same as the Tanenbaum Controls menu except there is no **Define Memory** selection. This is because only one register can be altered, the program counter (PC) register. The PC can be altered by choosing the **Update PC** selection from the **Registers** menu. The dialog box resulting from this selection is shown below:



Enter Desired Register Value:

```
0000000000001111
```

Using the various controls (as discussed in the Tanenbaum Example) the program is then run to completion. The memory contents after execution are shown below:

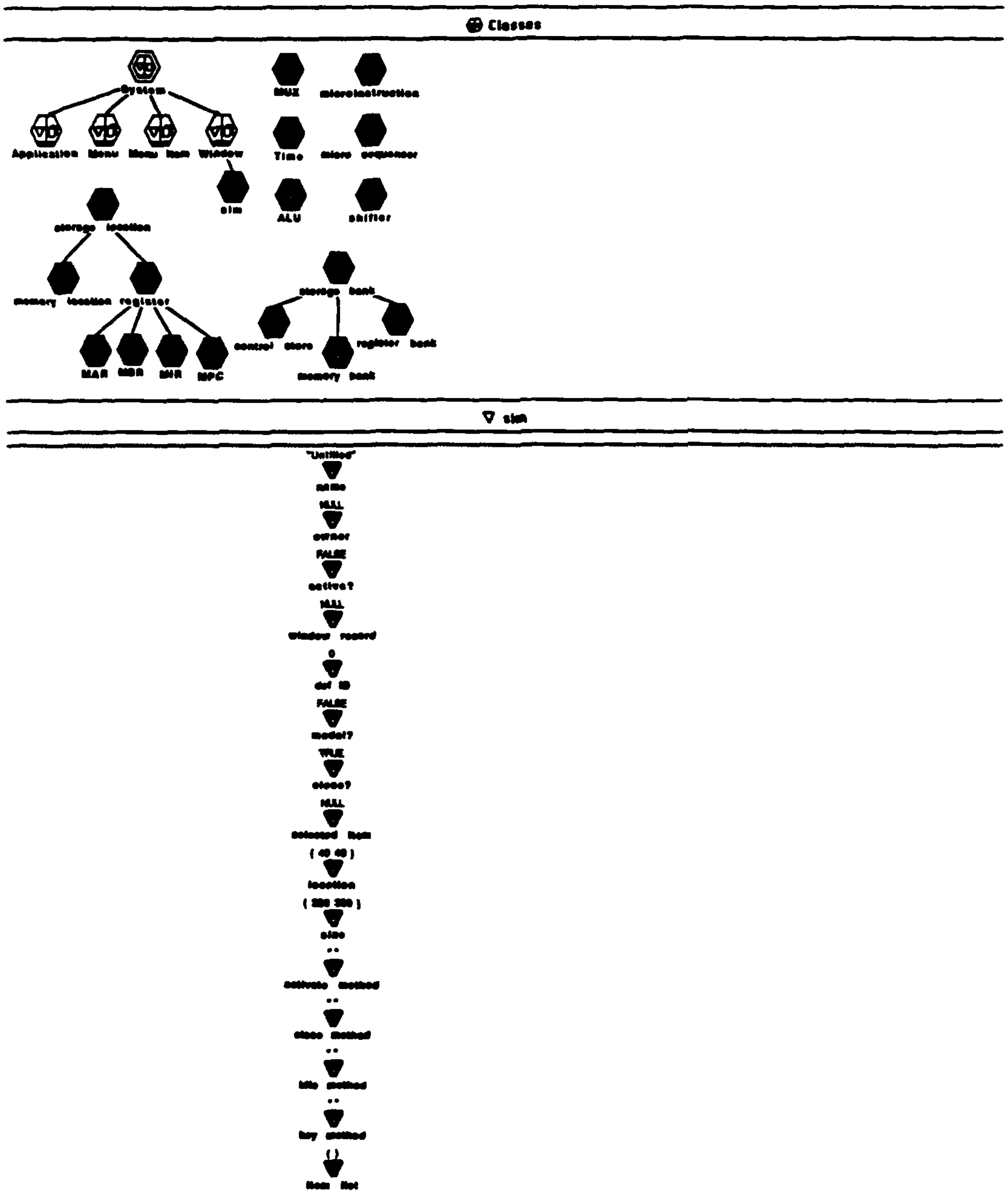


```
06--0000000000000000
07--0000000000000101
08--0000000000001111
09--0000000000001001
10--0000000000000000
11--0000000000001111
12--0000000000001111
13--0000000000001111
14--0000000000001111
15--0000000000001111
```

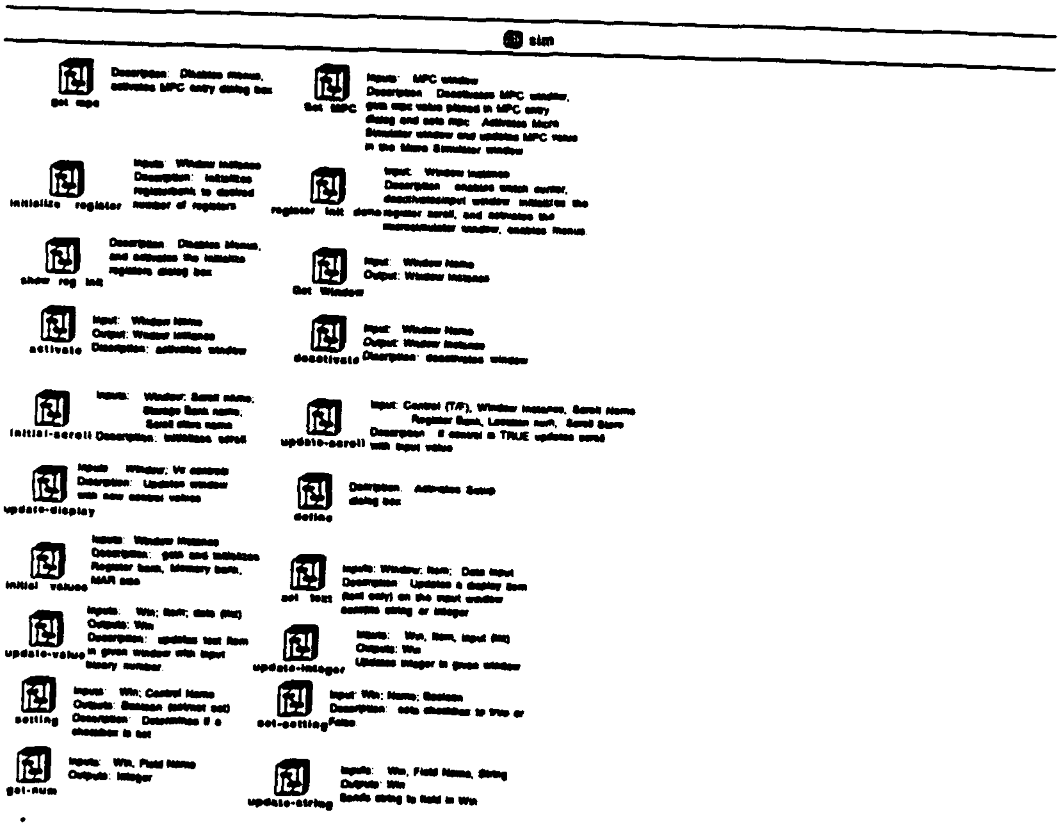
Memory locations 11-15 contain the value initially loaded into the accumulator 15. Memory location 11 is highlighted because it was the last memory value written to.

APPENDIX C

This appendix contains the source code for the Tanenbaum design microsimulator.



"Untitled"
name
NULL
owner
FALSE
active?
NULL
window record
0
def ID
FALSE
modal?
TRUE
close?
NULL
selected item
(40 40)
location
(256 256)
size
• •
activate method
• •
close method
• •
idle method
• •
key method
( )
item list



sim

get mpc — Description: Disables menus, activates MPC entry dialog box

Set MPC — Inputs: MPC window
Description: Deactivates MPC window, gets mpc value placed in MPC entry dialog and sets mpc. Activates Micro Simulator window and updates MPC value in the Micro Simulator window

initialize register — Inputs: Window instance
Description: Initializes registerbank to desired number of registers

register init done — Input: Window instance
Description: enables watch cursor, deactivates input window, initializes the register scroll, and activates the microsimulator window, enables menus.

show reg init — Description: Disables Menus, and activates the initialize registers dialog box

Get Window — Input: Window Name
Output: Window instance

activate — Input: Window Name
Output: Window instance
Description: activates window

deactivate — Input: Window Name
Output: Window instance
Description: deactivates window

initial-scroll — Inputs: Window; Scroll name; Storage Bank name; Scroll store name
Description: Initializes scroll

update-scroll — Input: Control (T/F), Window instance, Scroll Name, Register Bank, Location num, Scroll Store
Description: If control is TRUE updates scroll with input value

update-display — Inputs: Window; Vr controls
Description: Updates window with new control values

define — Description: Activates Setup dialog box

initial values — Inputs: Window instance
Description: gets and initializes Register bank, Memory bank, MAR etc

set text — Inputs: Window; Item; Data input
Description: Updates a display item (text only) on the input window accepts string or integer

update-value — Inputs: Win; Item; data (list)
Outputs: Win
Description: updates text item in given window with input binary number.

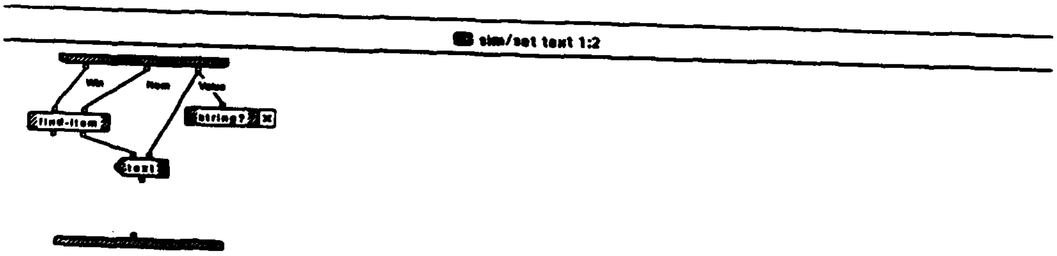
update-integer — Inputs: Win, Item, input (int)
Outputs: Win
Updates integer in given window

setting — Inputs: Win; Control Name
Outputs: Boolean (set/not set)
Description: Determines if a checkbox is set

set-setting — Input: Win; Name; Boolean
Description: sets checkbox to true or False

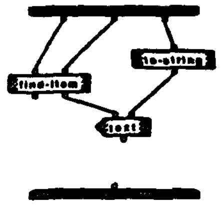
get-num — Inputs: Win, Field Name
Outputs: Integer

update-string — Inputs: Win, Field Name, String
Outputs: Win
Sends string to field in Win

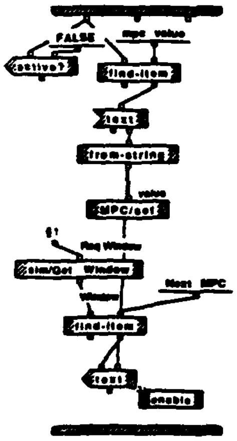


sim/set text 1:2

sim/set text 2:2

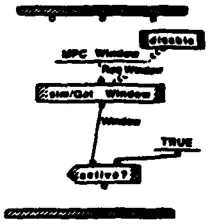


sim/Set MPC 1:1

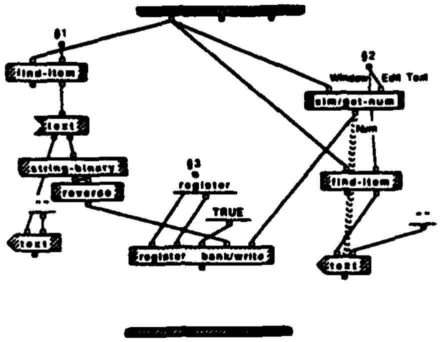


§1 Micro Simulator

sim/get mpc 1:1

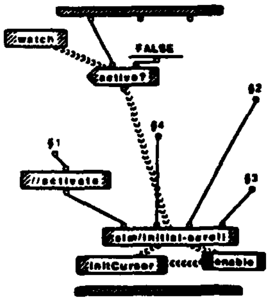


sim/initialize register 1:1



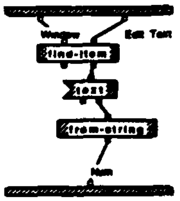
§1 register value
§2 register number
§3 register store

sim/register init done 1:1

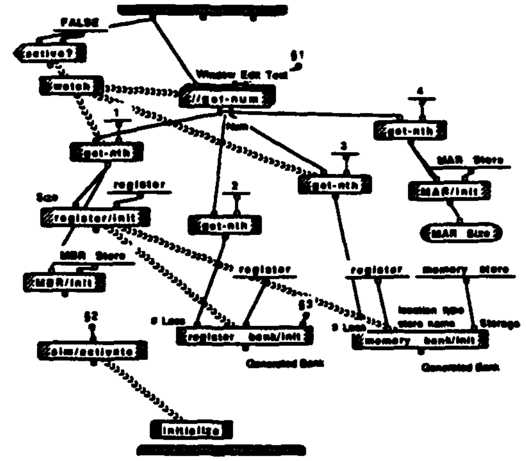


§1 Macro Simulator
§2 register store
§3 Register Scroll
§4 Register Values

sim/get-num 1:1

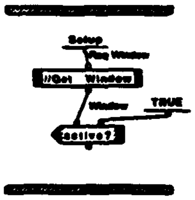


sim/initial values 1:1

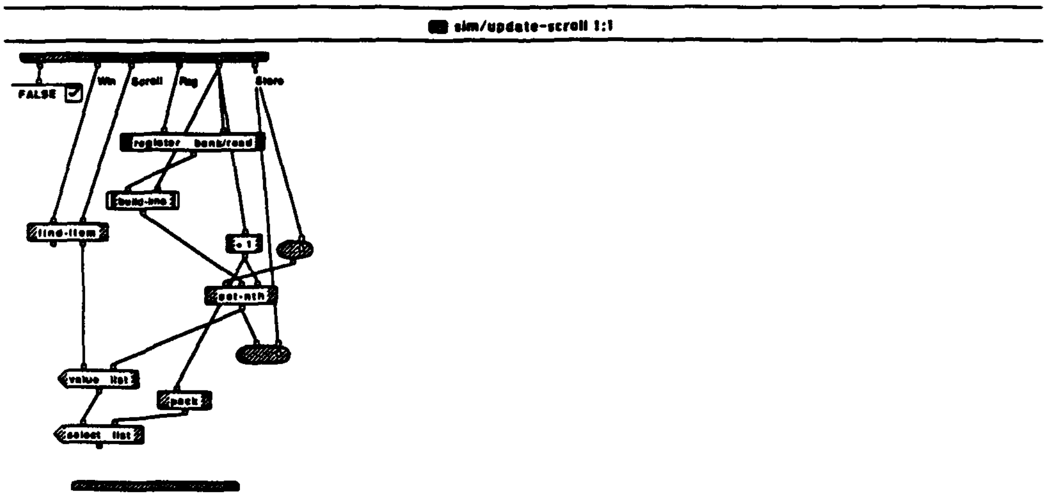


§1 (width registers memories mar)
§2 Micro Simulator
§3 register store

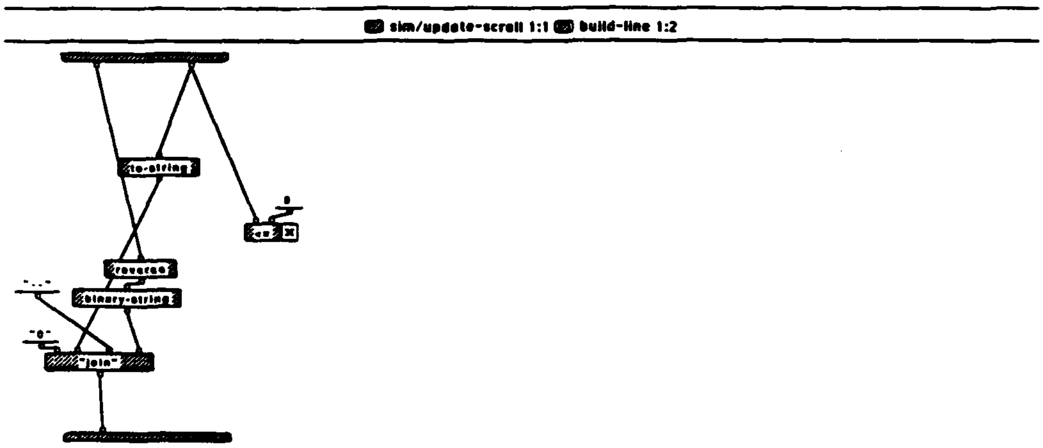
sim/define 1:1



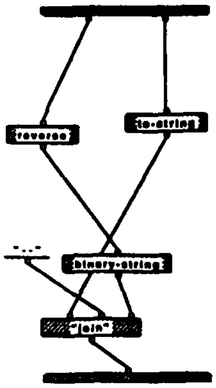
sim/update-scroll 1:1



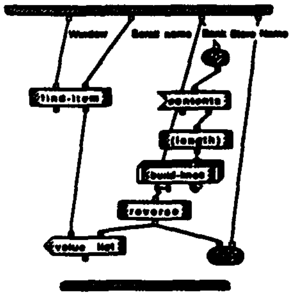
sim/update-scroll 1:1 build-line 1:2



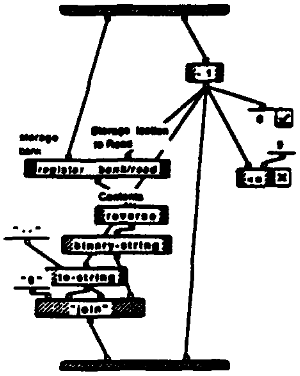
sim/update-scroll 1:1 build-line 2:2



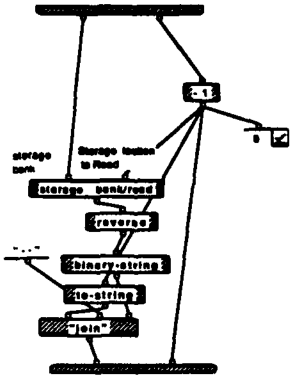
sim/initial-scroll 1:1



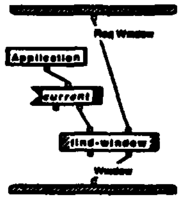
sim/initial-scroll 1:1 build-lines 1:2



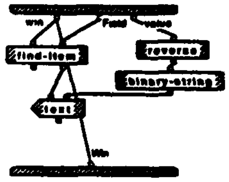
sim/initial-scroll 1:1 build-lines 2:2



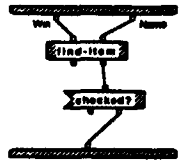
sim/Get Window 1:1



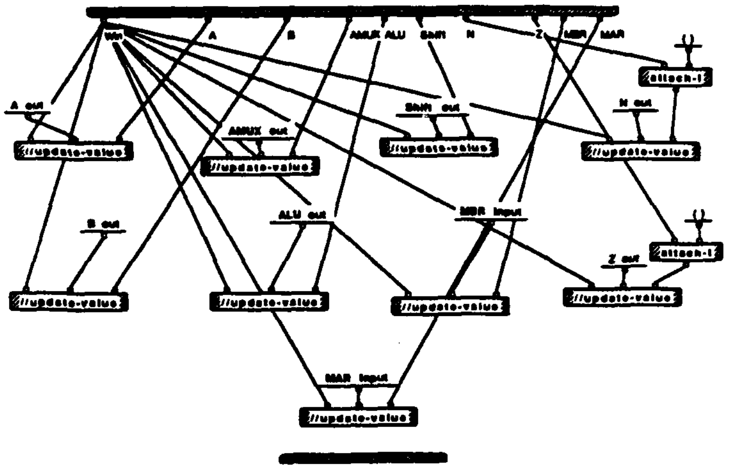
sim/update-value 1:1



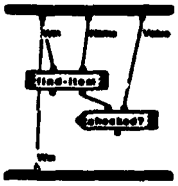
sim/setting 1:1



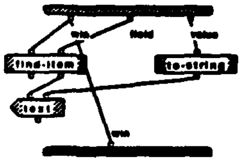
sim/update-display 1:1



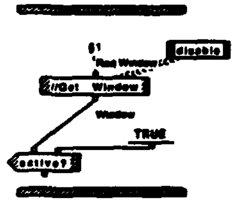
sim/set-setting 1:1



sim/update-integer 1:1



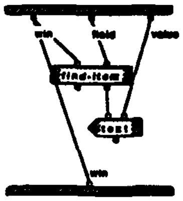
sim/show reg init 1:1



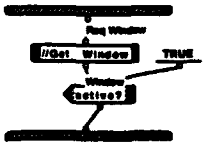
sim/show reg init 1:1

§1 Initialize Registers

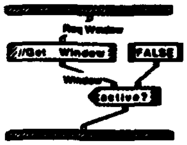
sim/update-string 1:1



sim/activate 1:1



sim/deactivate 1:1



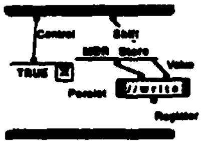
∇ MBR

MBR

mbr

Inputs Control (T/F); Bus
Outputs: None
Description: If control is TRUE writes bus input
into the MBR

MBR/mbr 1:1



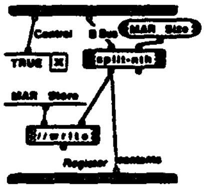
▽ MAR

MAR

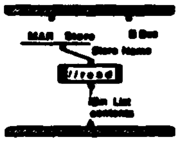
mar

Inputs: Control (T/F); Bus
Outputs: Bus
Description: If control is TRUE, the input is written into the MAR

MAR/mar 1:2



MAR/mar 2:2



▽ register

register

▽ storage location

( )
▼
contents

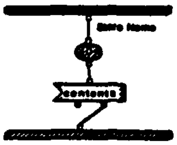
storage location

init
Inputs: Size; Store Name
Outputs: Storage Location instance
Description: Places an instance of Storage Location in specified Persistent

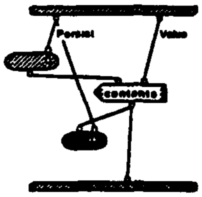
read
Inputs: Store Name
Outputs: Contents

write
Inputs: Store Name; Bus In
Outputs: Contents

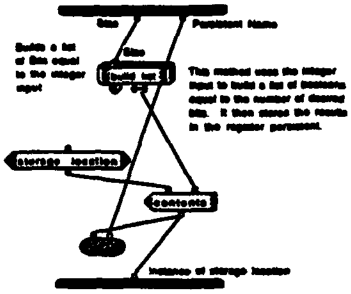
storage location/read 1:1



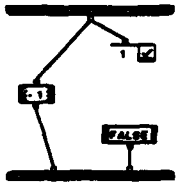
storage location/write 1:1



storage location/init 1:1


Size
Persistent Name
Builds a list
of Bits equal
to the integer
input
Size
build list
This method uses the integer
input to build a list of booleans
equal to the number of desired
bits. It then stores the results
in the register persistent.
storage location
contents
Instance of storage location

storage location/init 1:1 build list 1:1



▽ memory location

( )
▽
contents

memory location

▽ MIR

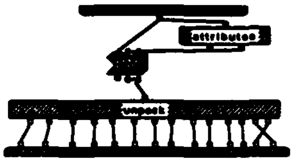
( )
▽
contents

MIR

decode

Input: Macroinstruction
Output: Control Signals
Description: Parses macroinstruction
into control signals

MIR/decode 1:1



▽ MPC

0
contents
0
cycles
0
counter

MPC

increment
Inputs: None
Outputs: Instance of MPC
Description: Increments cycle and counter variables. Instance is maintained in the MPC store.

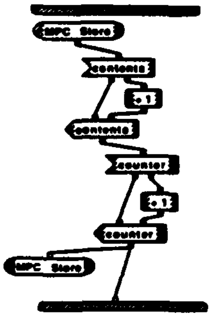
jump
Inputs: MPC In ...: Integer
Outputs: MPC Instance
Description: Sets contents of MPC instance to new value

set
Inputs: Bus (Integer)
Outputs: None
Description: Changes value of Contents of the MPC Instance residing in the MPC Store Persistent

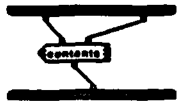
set cycles
Inputs: # of Cycles
Outputs: # of Cycles
Description: sets the cycles attribute to the input value. Resets counter attribute to 0. Uses the MPC Store Persistent

get counter
Inputs: None
Outputs: MPC counter value
Description: Uses MPC Store

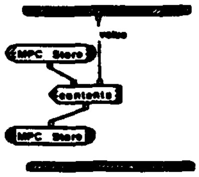
MPC/increment 1:1



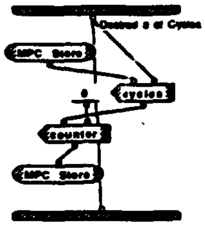
MPC/jump 1:1



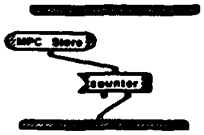
MPC/set 1:1



MPC/set cycles 1:1



MPC/get counter 1:1



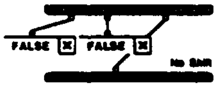
▽ shifter

shifter

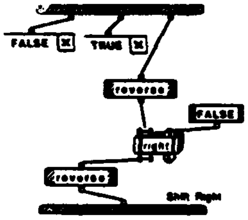
shifter

input: two control signals and
position list
output: result (left, right, nodiff)

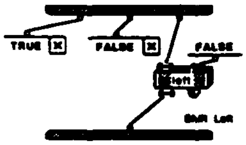
shifter/shifter 1:4



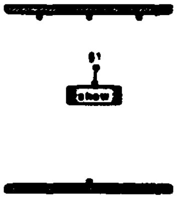
shifter/shifter 2:4



shifter/shifter 3:4

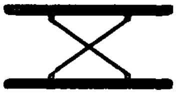


shifter/shifter 4:4

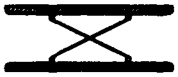


§1 Error in ALU/shifter

shifter/shifter 2:4 right 1:1



shifter/shifter 3:4 left 1:1



▽ ALU

ALU

bit not — Input: Bin list
output: not of Bin list

bit add — Input: Two bin lists
output: sum of bin list

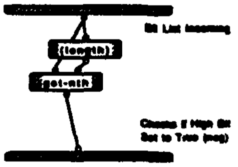
math — Performs: add, and, A, not
outputs: result, Z, N

high bit — Input: list of boolean
output: high order bit

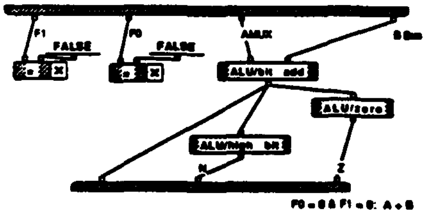
zero — Input: boolean list
output: true if zero
false if not zero

bit and — Input: two Bin lists
output: 1 bin list (and of A & B)

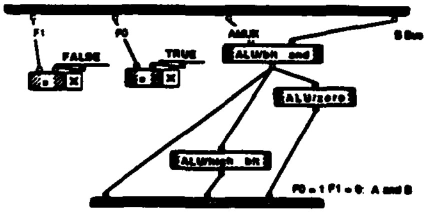
ALU/high bit 1:1



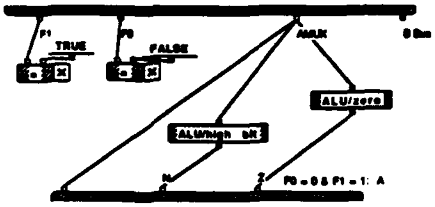
ALU/math 1:5



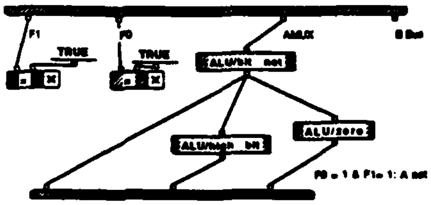
ALU/math 2:5



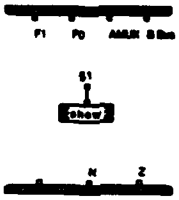
ALU/math 3:5



ALU/math 4:5

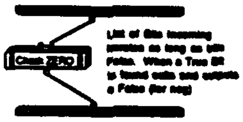


ALU/math 5:5

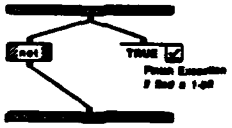


§1. Error in ALU/math

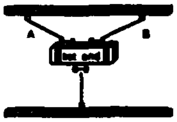
ALU/zero 1:1



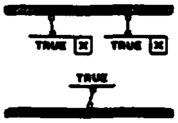
ALU/zero 1:1 Check ZERO 1:1



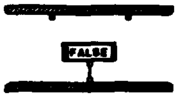
ALU/bit and 1:1



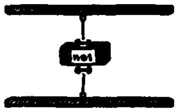
ALU/bit and 1:1 list and 1:2



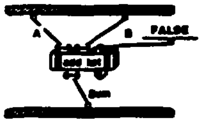
ALU/bit and 1:1 list and 2:2



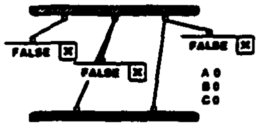
ALU/bit not 1:1



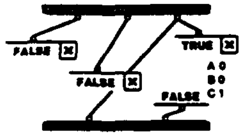
ALU/bit add 1:1



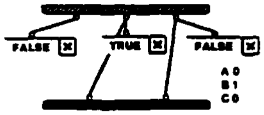
ALU/bit add 1:1 add list 1:9



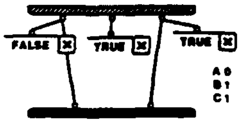
▨ ALU/bit add 1:1 ▨ add list 2:9



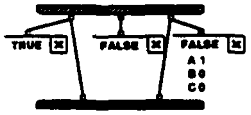
▨ ALU/bit add 1:1 ▨ add list 3:9



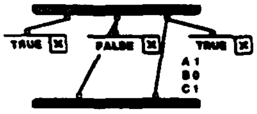
▨ ALU/bit add 1:1 ▨ add list 4:9



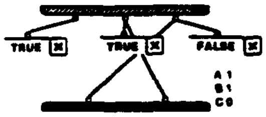
▨ ALU/bit add 1:1 ▨ add list 5:9



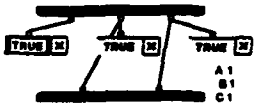
▨ ALU/bit add 1:1 ▨ add list 6:9



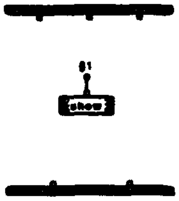
▨ ALU/bit add 1:1 ▨ add list 7:9



ALU/bit add 1:1 add list 8:9



ALU/bit add 1:1 add list 9:9



§1 error in add list

▽ microinstruction

FALSE
▽
AMUX

(FALSE FALS...
▽
COND

(FALSE FALS...
▽
ALU

(FALSE FALS...
▽
SH

FALSE
▽
MBR

FALSE
▽
MAR

FALSE
▽
RD

FALSE
▽
WR

FALSE
▽
ENC

(FALSE FALS...
▽
C

(FALSE FALS...
▽
B

(FALSE FALS...
▽
A

0
▽
ADDR

NULL
▼
memo

∇ control store

( )
contents

control store

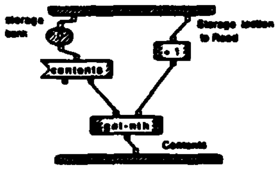
init — Input: Store Name
Generates empty instance
of c.s. and loads into persistent

load — Inputs: Store Name

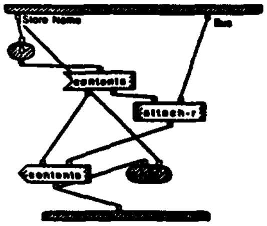
read — Input: Store Name; Int Location
Output: Contents

add location — Inputs: Store Name, Bus
Outputs: Revised Control Store Contents
Descript: Adds a new Microinstruction to
the Control Store contents.

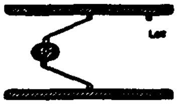
control store/read 1:1



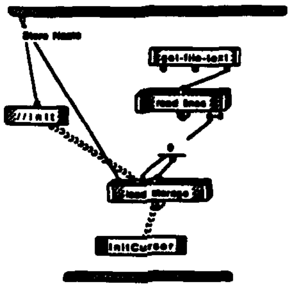
control store/add location 1:2



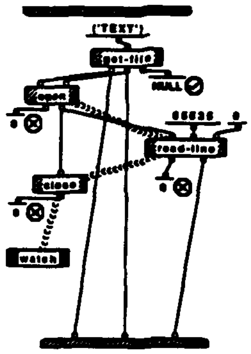
control store/add location 2:2



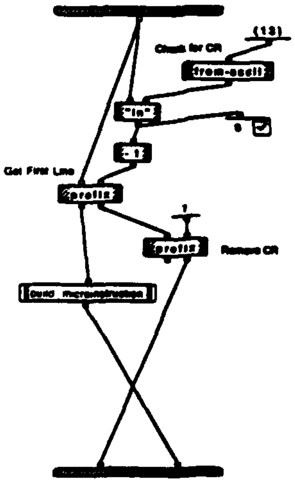
control store/load 1:1



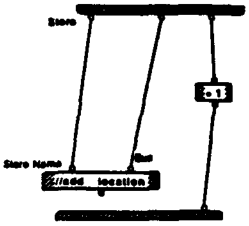
control store/load 1:1 get-file-text 1:1



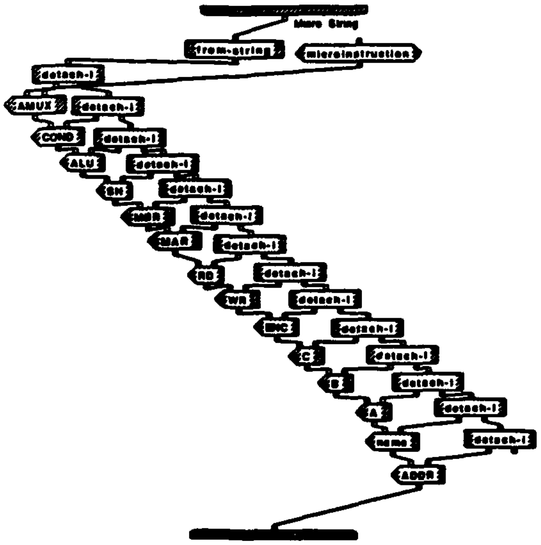
control store/load 1:1 read lines 1:1



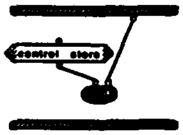
▨ control store/load 1:1 ▨ load storage 1:1



▨ control store/load 1:1 ▨ read lines 1:1 ▨ build microinstruction 1:1



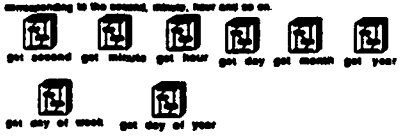
▨ control store/init 1:1



∇ Time

Time

The following methods all return an integer value corresponding to the second, minute, hour and so on.



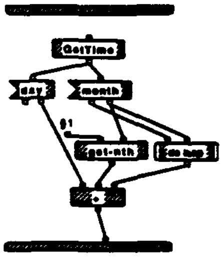
The following methods all return a string.

get day name
get month name
get time
get date

includes seconds if input is TRUE.

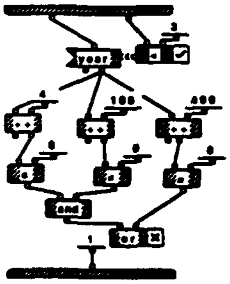
short = 0
long = 1
abbrv. long =2

Time/get day of year 1:1

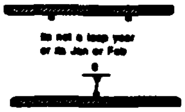


§1 (0 31 59 90 120 151 181 212 243 273 304 334 365)

Time/get day of year 1:1 do leap 1:2

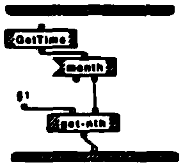


Time/get day of year 1:1 do leap 2:2



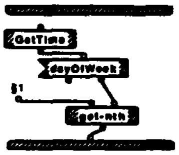
its not a leap year
or its Jan or Feb

Time/get month name 1:1



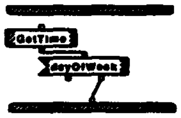
§1 (January February March April May June July August September October November December)

Time/get day name 1:1

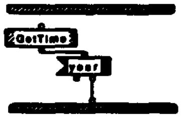


§1 (Sunday Monday TuesDay Wednesday Thursday Friday Saturday)

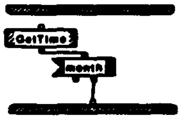
Time/get day of week 1:1



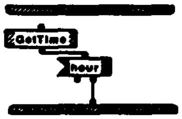
Time/get year 1:1



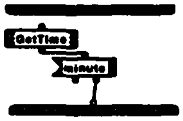
Time/get month 1:1



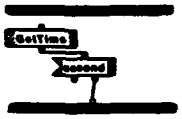
Time/get hour 1:1



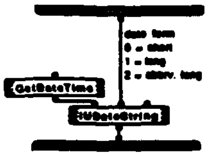
Time/get minute 1:1



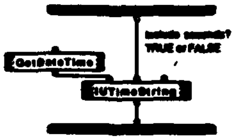
Time/get second 1:1



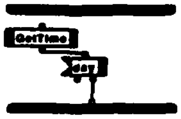
Time/get date 1:1



Time/get time 1:1



Time/get day 1:1



▽ storage bank

storage bank

init — Inputs: #Locs; Location type; Storage person
Outputs: Generated Bank

read — Inputs: Storage Bank; Location #
Outputs: Location Contents

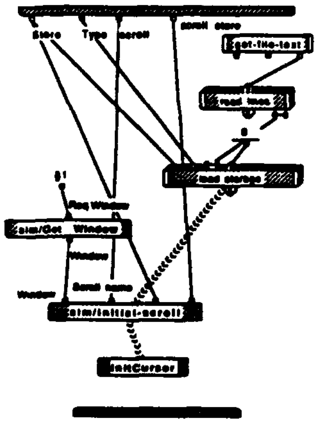
load — Inputs: Store; Type; Scroll, Scroll Store
Outputs: None

write — Inputs: Store Name; Location Type,
Control; Bus; Loc
Outputs: Contents

BIN-read — Inputs: Binary Location; Location Store
Outputs: Contents

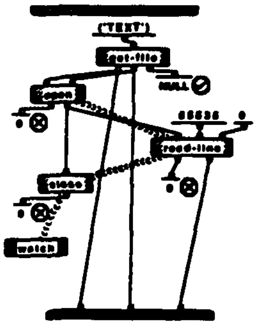
BIN-write — Inputs: Store Name; Location Type,
Control; Bus In, Bin Location
Outputs: Contents, Integer Location

storage bank/load 1:1

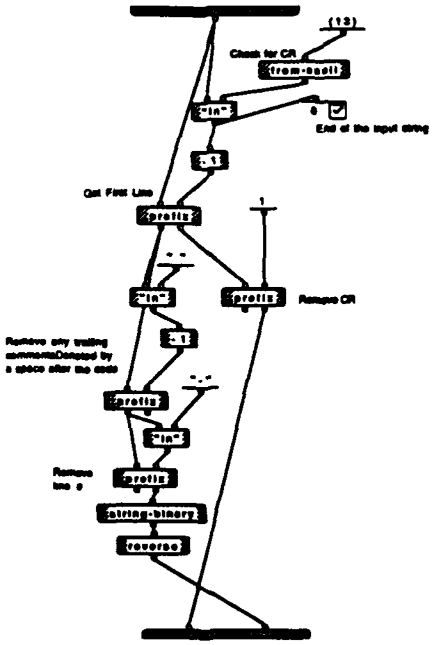


§1 Micro Simulator

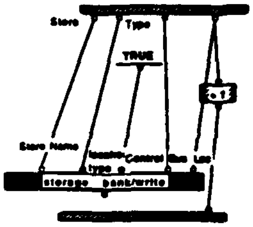
storage bank/load 1:1 get-file-text 1:1



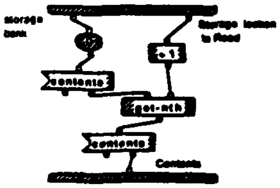
storage bank/load 1:1 read lines 1:1



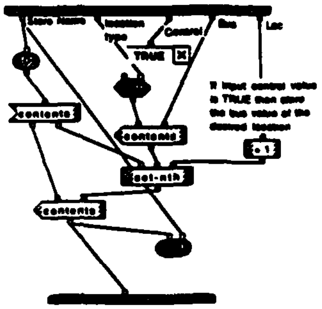
storage bank/load 1:1 load storage 1:1



storage bank/read 1:1

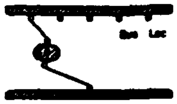


storage bank/write 1:2

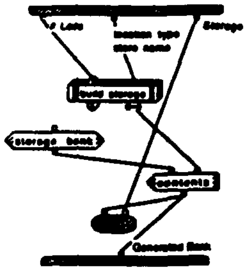


If input control value is TRUE then store the bus value of the desired location

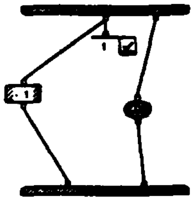
storage bank/write 2:2



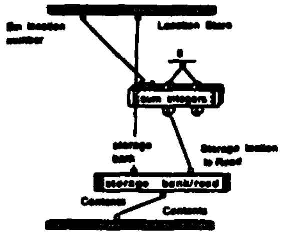
storage bank/init 1:1



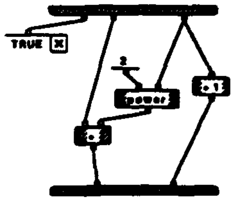
storage bank/init 1:1 build storage 1:1



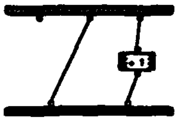
storage bank/BIN-read 1:1



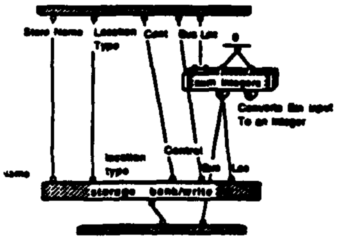
storage bank/BIN-read 1:1 sum integers 1:2



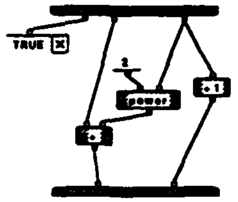
storage bank/BIN-read 1:1 sum integers 2:2



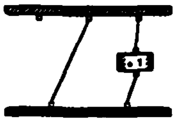
storage bank/BIN-write 1:1



storage bank/BIN-write 1:1 sum integers 1:2



storage bank/BIN-write 1:1 sum integers 2:2



∇ memory bank

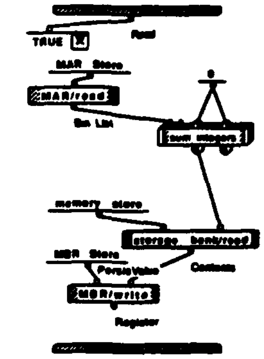
memory bank

load — Description: supplies appropriate inputs to superclasses load method for loading the memory bank.

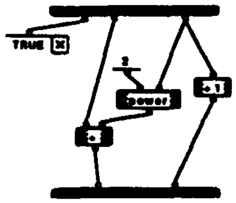
write — Inputs: Bus input; Write Control (T/F)
Outputs: Location Value; Int Location

read — Inputs: Control (T/F) to enable read
Description: Uses MAR contents as location to read memory. Writes results to MBR

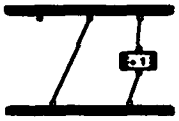
memory bank/read 1:1



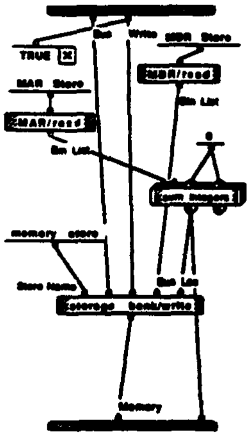
▬ memory bank/read 1:1 ▬ sum integers 1:2



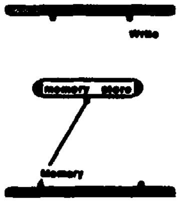
▬ memory bank/read 1:1 ▬ sum integers 2:2



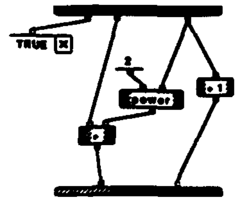
▨ memory bank/write 1:2



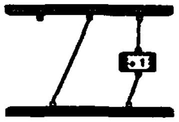
▬ memory bank/write 2:2



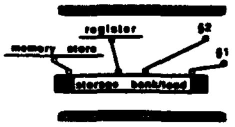
memory bank/write 1:2 sum integers 1:2



memory bank/write 1:2 sum integers 2:2



memory bank/load 1:1



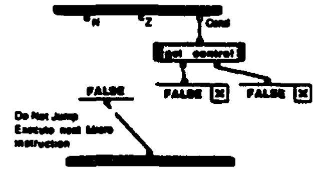
§1 Memory Scroll
§2 Memory Values

∇ micro sequencer

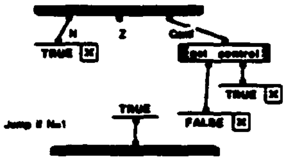
micro sequencer

Input: N,Z,Cond
Output: Boolean Control Signal

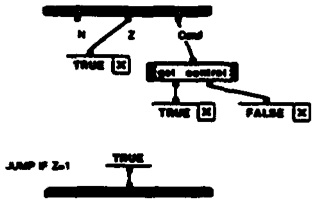
micro sequencer/generate signal 1:5



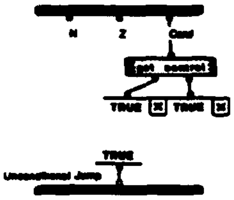
micro sequencer/generate signal 2:5



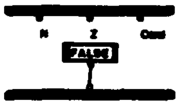
micro sequencer/generate signal 3:5



micro sequencer/generate signal 4:5



micro sequencer/generate signal 5:5



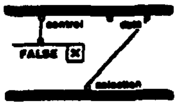
▽ MUX

MUX

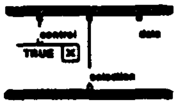
MUX

Input: Control, Data1 & Data2
Control=0->Select Right
Control=1->Select Left

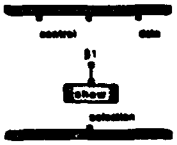
MUX/mux 1:3



MUX/mux 2:3



MUX/mux 3:3



§1. MBR Control Data Input Error

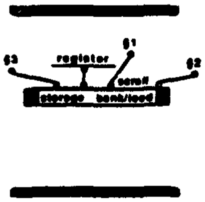
∇ register bank

register bank

load

Description: supplies appropriate inputs to superclasses load method for loading the register bank.

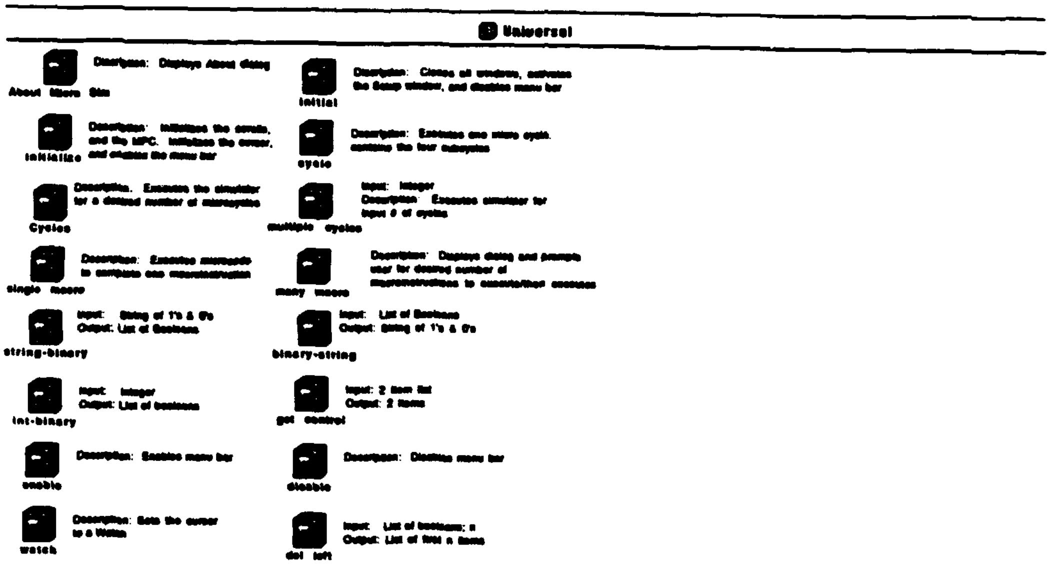
register bank/load 1:1



§1 Register Values

■ register bank/load 1:1

§2 Register load
§3 register store



Universal

About Micro Sim
Description: Displays About dialog

initial
Description: Closes all windows, activates the Setup window, and disables menu bar

initialize
Description: Initializes the screen, and the MPC. Initializes the cursor, and enables the menu bar

cycle
Description: Executes one micro cycle, contains the four subcycles

Cycles
Description. Executes the simulator for a desired number of microcycles

multiple cycles
Input: Integer
Description: Executes simulator for input # of cycles

single macro
Description: Executes microcode to complete one macroinstruction

many macro
Description: Displays dialog and prompts user for desired number of macroinstructions to execute/then executes

string-binary
Input: String of 1's & 0's
Output: List of Booleans

binary-string
Input: List of Booleans
Output: String of 1's & 0's

int-binary
Input: Integer
Output: List of booleans

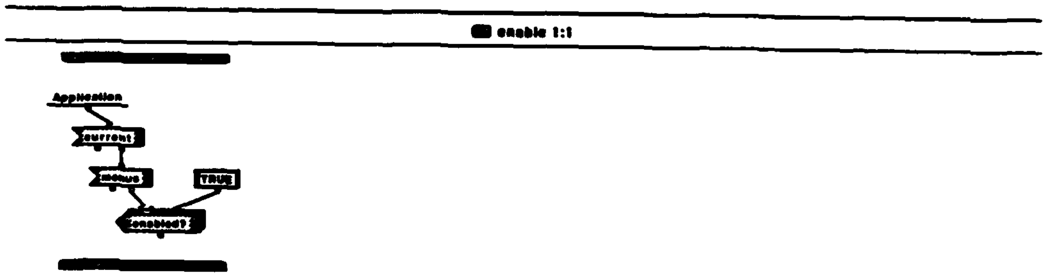
get control
Input: 2 item list
Output: 2 items

enable
Description: Enables menu bar

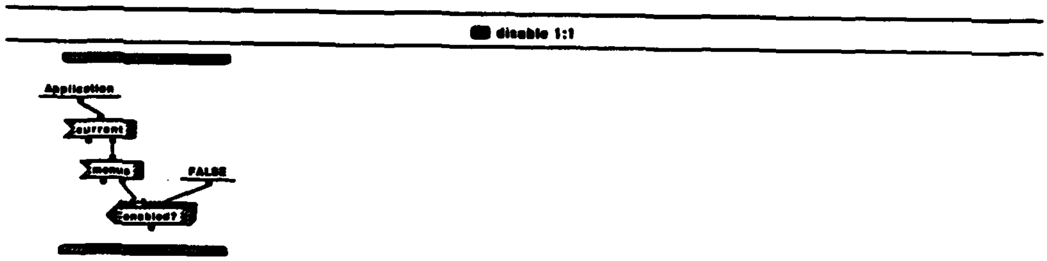
disable
Description: Disables menu bar

watch
Description: Sets the cursor to a Watch

del left
Input: List of booleans; n
Output: List of first n items

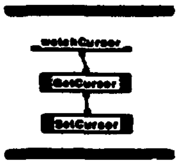


enable 1:1

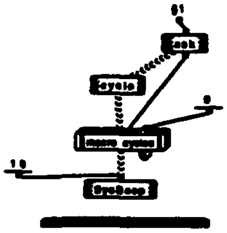


disable 1:1

watch 1:1

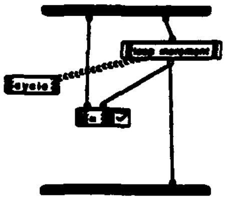


many macro 1:1

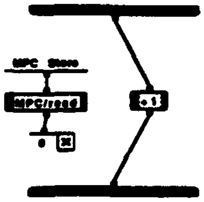


§1. Enter desired number of Macro instructions to execute:

many macro 1:1 macro cycles 1:1

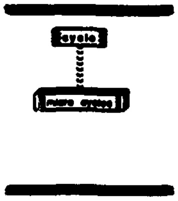


many macro 1:1 macro cycles 1:1 loop increment 1:2

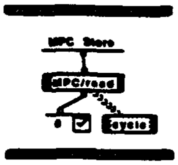


many macro 1:1 macro cycles 1:1 loop increment 2:2

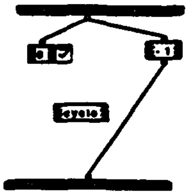
single macro 1:1



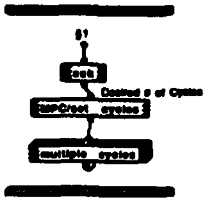
single macro 1:1 micro cycles 1:1



multiple cycles 1:1

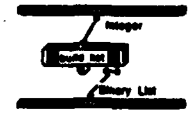


Cycles 1:1

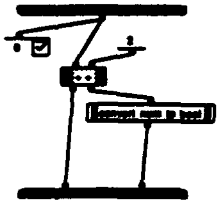


§1 Enter Desired # of Machine Cycles to Run:

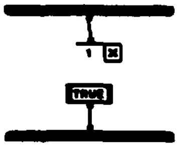
int-binary 1:1



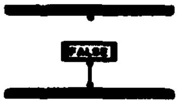
int-binary 1:1 build list 1:1



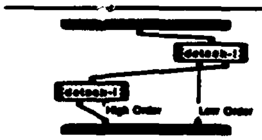
int-binary 1:1 build list 1:1 convert num to bool 1:2



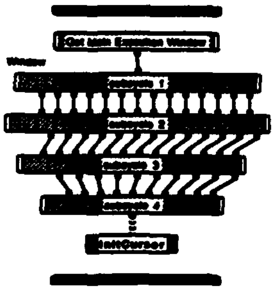
int-binary 1:1 build list 1:1 convert num to bool 2:2



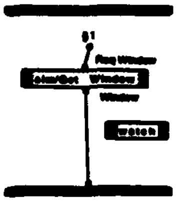
get control 1:1



cycle 1:1

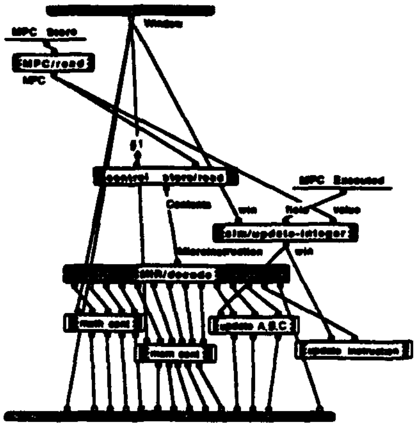


cycle 1:1 Get Main Execution Window 1:1



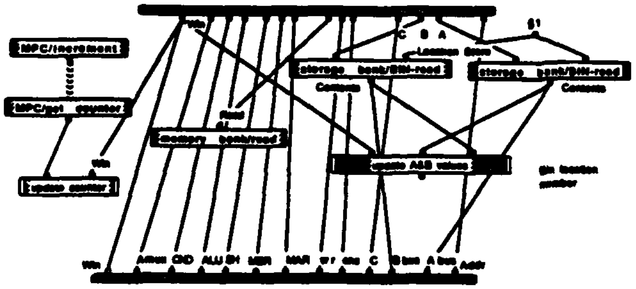
§1 Micro Simulator

cycle 1:1 subcycle 1 1:1



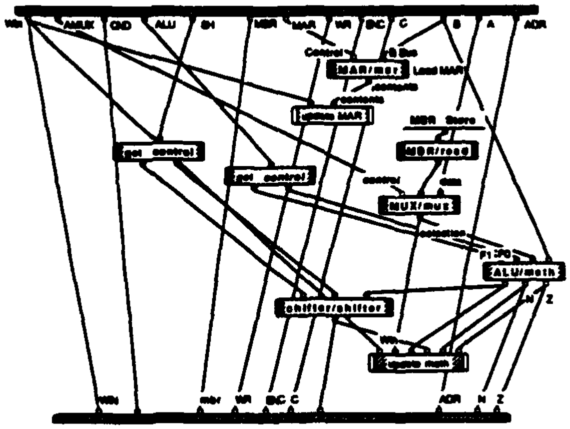
§1 control store

cycle 1:1 subcycle 2 1:1

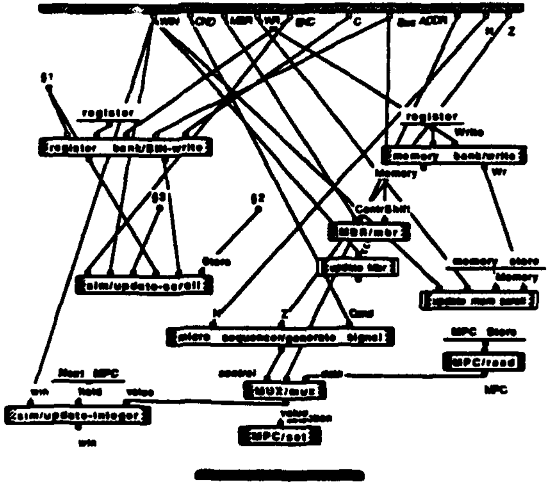


§1 register store

cycle 1:1 subcycle 3 1:1

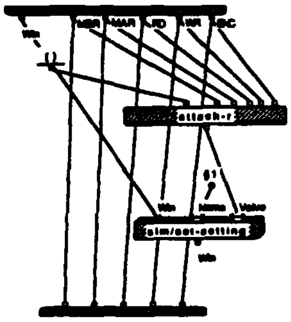


cycle 1:1 subcycle 4 1:1



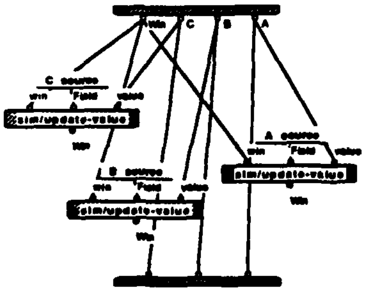
§1 register store
§2 Register Scroll
§3 Register Values

cycle 1:1 subcycle 1 1:1 mem cont 1:1

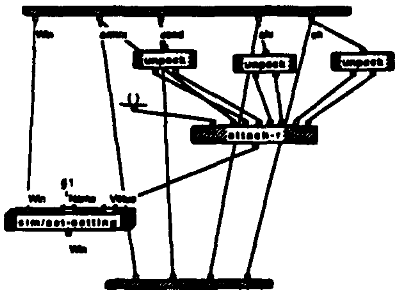


§1 (MBR MAR Read Write ENC)

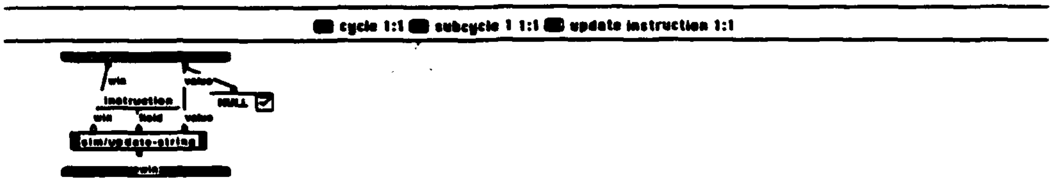
cycle 1:1 subcycle 1 1:1 update A,B,C 1:1



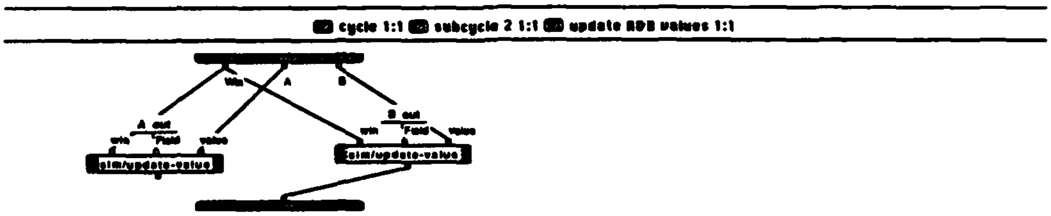
cycle 1:1 subcycle 1 1:1 math cont 1:1



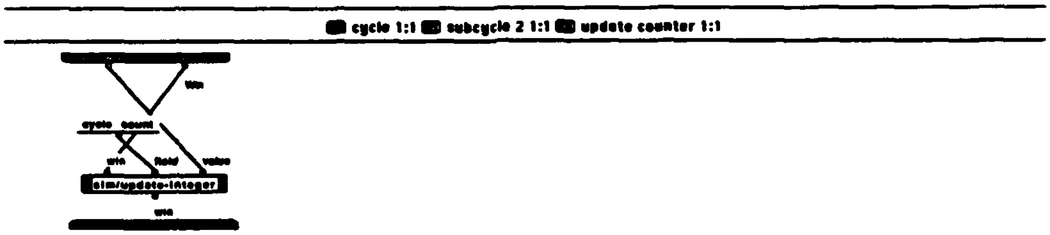
§1 (A0 C0 C1 F0 F1 H0 H1)



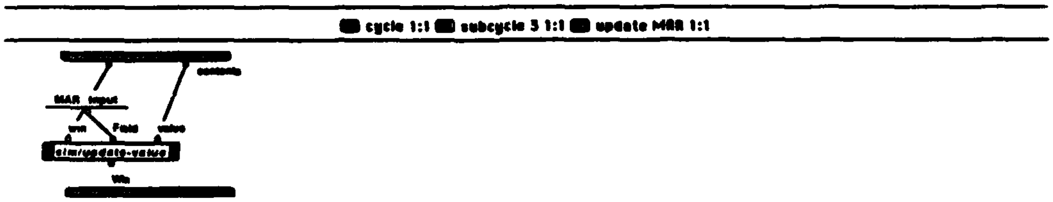
cycle 1:1 subcycle 1 1:1 update instruction 1:1



cycle 1:1 subcycle 2 1:1 update RDB values 1:1

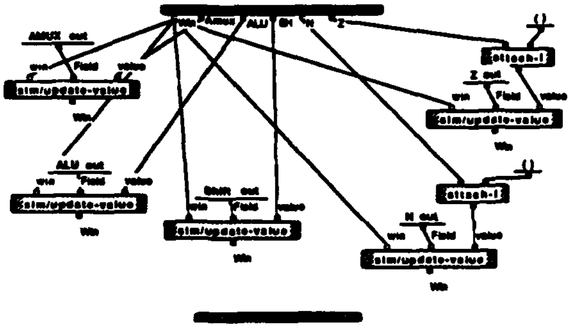


cycle 1:1 subcycle 2 1:1 update counter 1:1

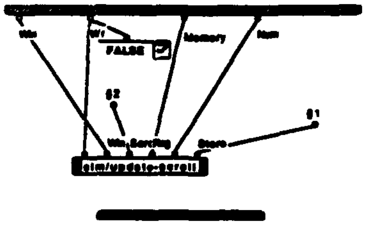


cycle 1:1 subcycle 3 1:1 update MAR 1:1

■ cycle 1:1 ■ subcycle 3 1:1 ■ update math 1:1

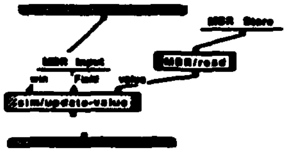


■ cycle 1:1 ■ subcycle 4 1:1 ■ update mem scroll 1:1

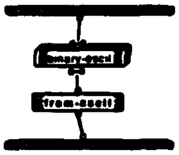


§1 Memory Scroll
§2 Memory Values

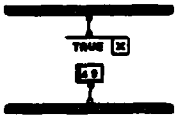
■ cycle 1:1 ■ subcycle 4 1:1 ■ update Mbr 1:1



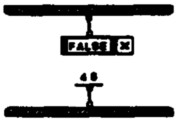
■ binary-string 1:1



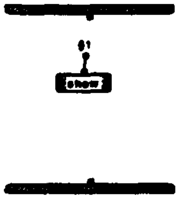
binary-string 1:1 binary-ascii 1:3



binary-string 1:1 binary-ascii 2:3

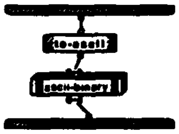


binary-string 1:1 binary-ascii 3:3

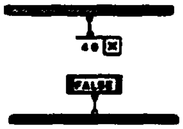


§1 error in binary-ascii

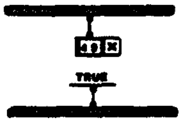
string-binary 1:1



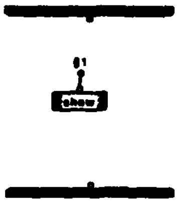
string-binary 1:1 ascii-binary 1:3



string-binary 1:1 ascii-binary 2:3

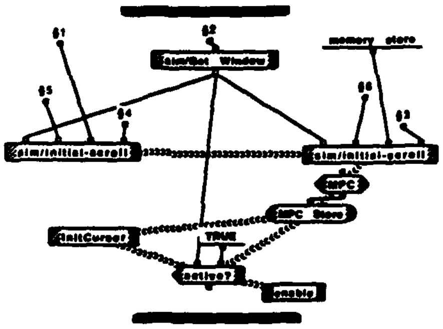


string-binary 1:1 ascii-binary 3:5



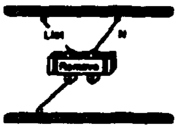
§1 error in ascii-binary

initialize 1:1

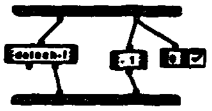


§1 register store
§2 Micro Simulator
§3 Memory Scroll
§4 Register Scroll
§5 Register Values
§6 Memory Values

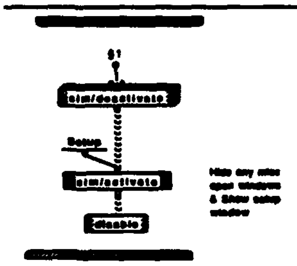
del left 1:1



del left 1:1 Remove 1:1

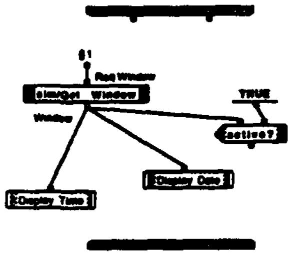


Initial 1:1



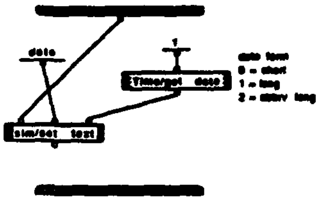
$1 ("Micro Simulator" "Setup" "Initialize Registers" "MPC Window" "About NPS Micro Simulator")

About Micro Sim 1:1

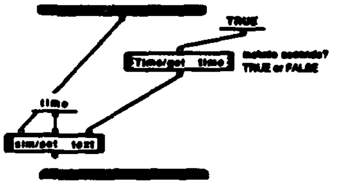


$1 About NPS Micro Simulator

About Micro Sim 1:1 Display Date 1:1



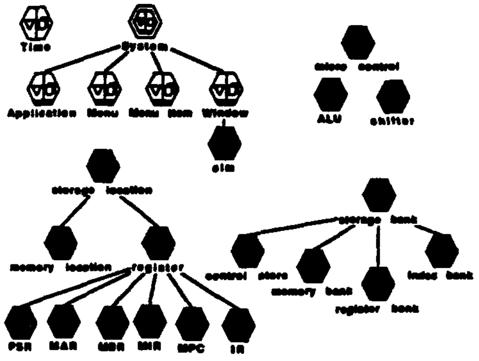
About Micro Sim 1:1 Display Time 1:1



APPENDIX D

This appendix contains the source code for the ASC design microsimulator.

Classes



▽ PSR

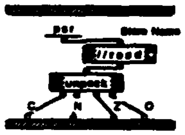
contents

PSR

decode

Outputs: C; N; Z; O
Description: uses psr store to output contents of psr

PSR/decode 1:1



▽ MBR

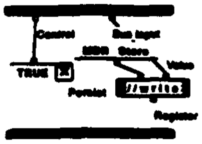
contents

MBR

mbr

Inputs: Control (T/F); Bus input
Outputs: None
Description: If control is TRUE writes bus input into the MBR

MBR/mbr 1:1



▽ MAR

()
contents

MAR

▽ register

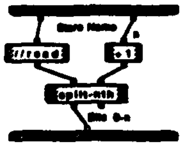
()
contents

register

read low

Inputs: Store Name; n
Output: Bits 0-n

register/read low 1:1



▽ storage location

()
contents

storage location

init

Inputs: Size; Store Name
Outputs: Storage Location Instance
Description: Places an instance of
Storage Location in specified Persistent

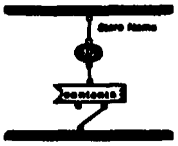
read

Inputs: Store Name
Outputs: Contents

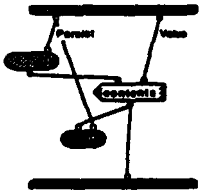
write

Inputs: Store Name; Bus n
Outputs: Contents

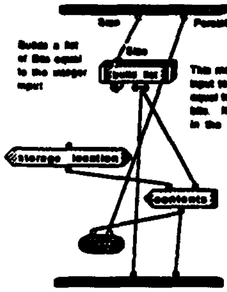
storage location/read 1:1



storage location/write 1:1



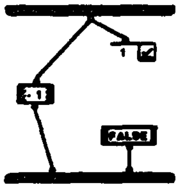
storage location/init 1:1



Builds a list of Bits equal to the integer input

This method uses the integer input to build a list of booleans equal to the number of desired bits. It then stores the results in the register persistent.

storage location/init 1:1 build list 1:1



▽ memory location

contents

■ memory location

▽ MIR

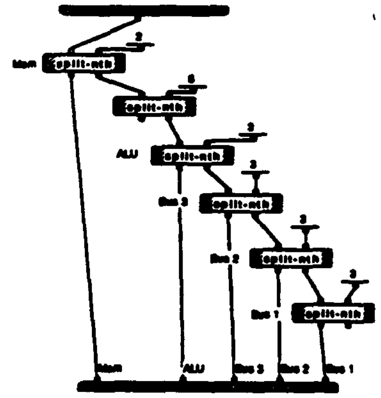
contents

■ MIR

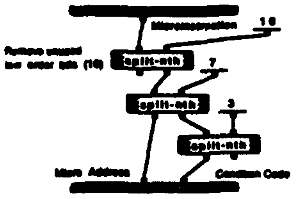
decode 0
Inputs: Control Store Loc
Outputs: MEM; ALU; BUS3; BUS2; BUS1
Description: outputs type zero control signals

decode 1
Inputs: Control Store Loc
Outputs: Micro Address; Condition Code
Description: outputs type one signals

■ MIR/decode 0 1:1



MIR/decode 1 1:1



▽ MPC

32
▽
contents

0
▽
cycles

0
▽
counter

MPC

increment
Outputs: <<MPC>>
Description: Uses parallel name MPC Store. Increments counter & contents

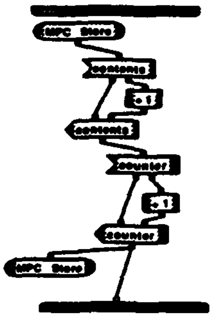
get counter
Outputs: Counter Contents
Description: Uses MPC Store to return contents of counter

jump
Inputs: <<MPC>>; Bus in
Outputs: <<MPC>>
Description: Resets contents attribute

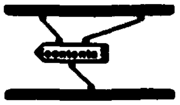
set
Inputs: Bus in
Description: Uses MPC Store to update contents attribute

set cycles
Inputs: Desired # of Cycles
Description: Uses MPC Stee Parallel resets counter=0 and sets cycles attribute

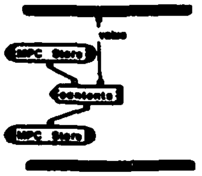
MPC/increment 1:1



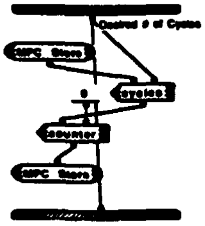
MPC/jump 1:1



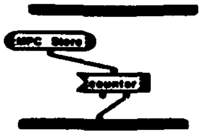
MPC/set 1:1

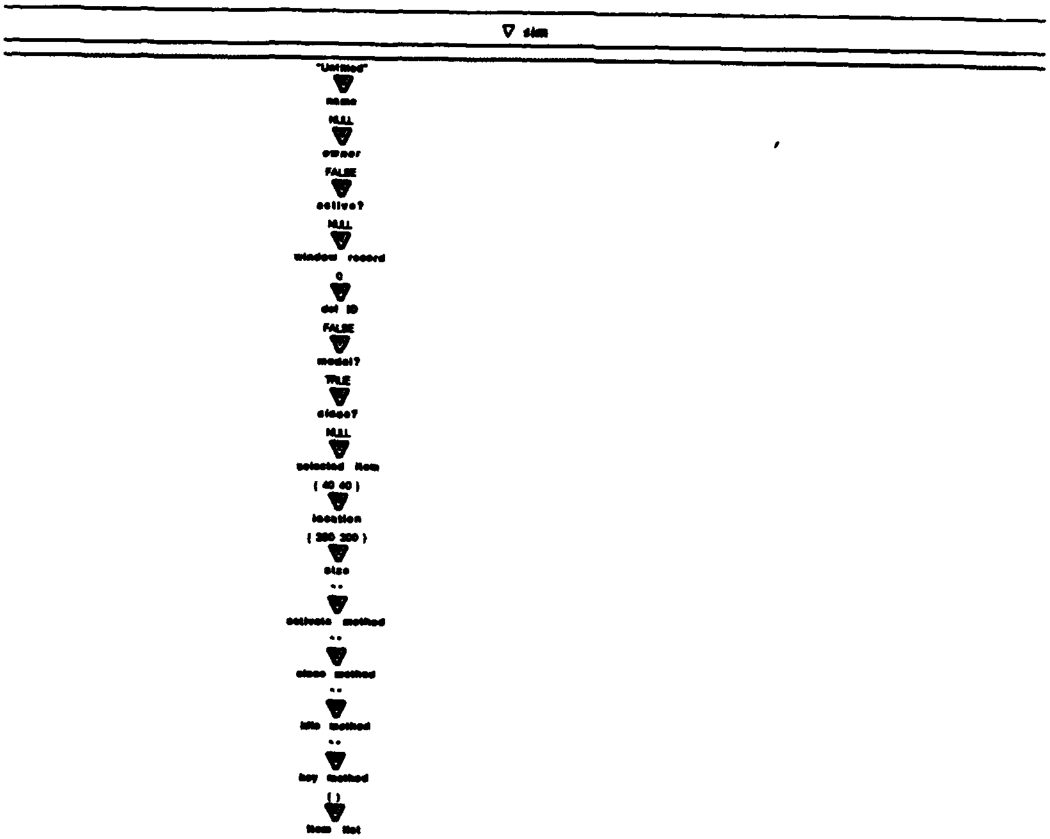


MPC/set cycles 1:1



MPC/get counter 1:1





∇ sim

"Untitled"
name

NULL
owner

FALSE
active?

NULL
window record

0
dlg ID

FALSE
modal?

TRUE
close?

NULL
selected item

(40 40)
location

(200 200)
size

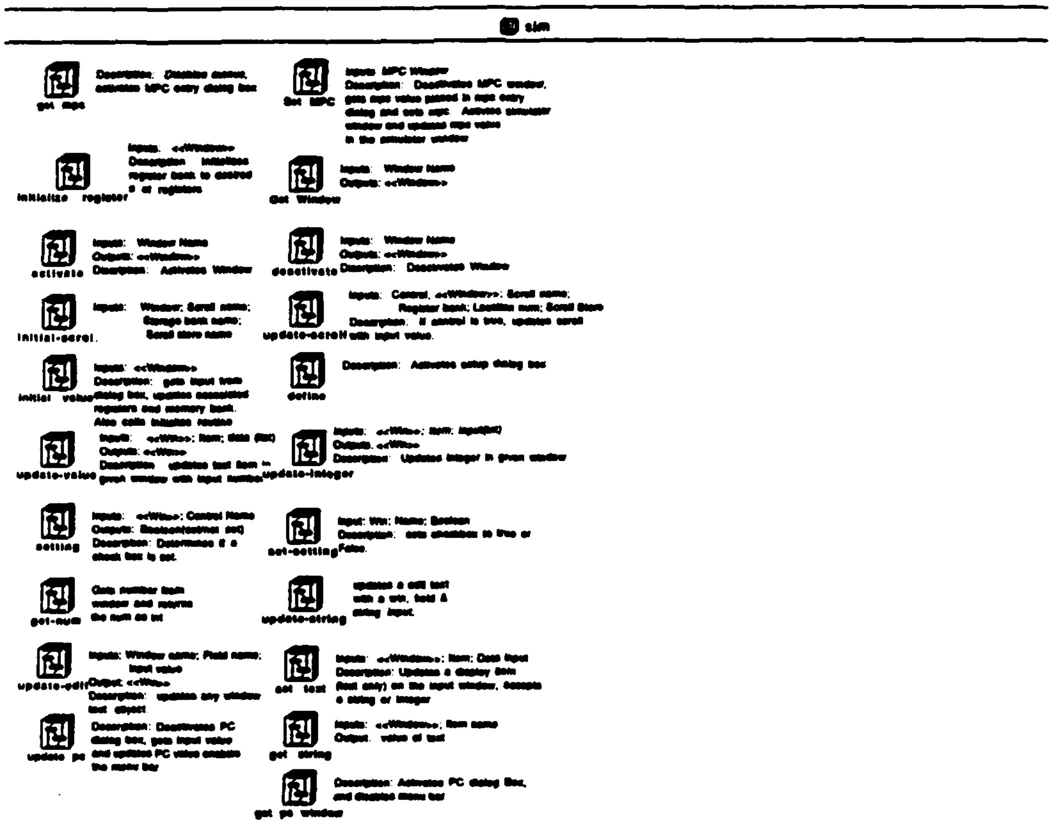
• •
activate method

• •
close method

• •
idle method

• •
key method

( )
item list



sim

get mpc
Description: Disables menus, activates MPC entry dialog box

Set MPC
Inputs: MPC Window
Description: Deactivates MPC window, gets mpc value placed in mpc entry dialog and sets mpc. Activate simulator window and updates mpc value in the simulator window

initialize register
Inputs: <<Window>>
Description: Initializes register bank to desired # of registers

Get Window
Inputs: Window Name
Outputs: <<Window>>

activate
Inputs: Window Name
Outputs: <<Window>>
Description: Activates Window

deactivate
Inputs: Window Name
Outputs: <<Window>>
Description: Deactivates Window

initial-scrol.
Inputs: Window; Scroll name; Storage bank name; Scroll store name

update-scroll
Inputs: Control, <<Window>>; Scroll name; Register bank; Location num; Scroll Store
Description: If control is true, updates scroll with input value.

initial value
Inputs: <<Window>>
Description: gets input from dialog box, updates associated registers and memory bank. Also calls intialize routine

define
Description: Activates setup dialog box

update-value
Inputs: <<Win>>; Item; data (list)
Outputs: <<Win>>
Description: updates text item in given window with input number

update-integer
Inputs: <<Win>>; item; input(int)
Outputs. <<Win>>
Description: Updates integer in given window

setting
Inputs: <<Win>>; Control Name
Outputs: Boolean(set/not set)
Description: Determines if a check box is set.

set-setting
Input: Win; Name; Boolean
Description: sets checkbox to true or False.

get-num
Gets number from window and returns the num as int

update-string
updates a edit text with a win, field & string input.

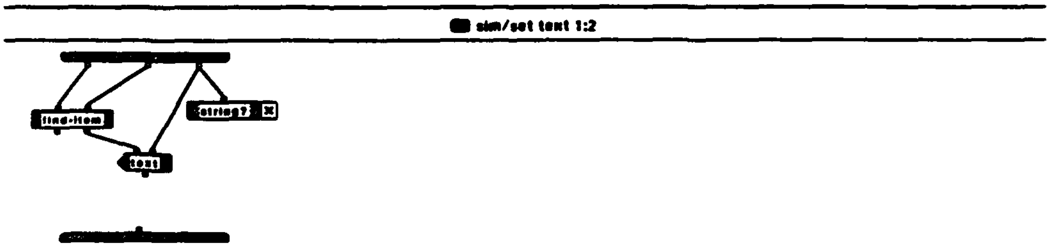
update-edit
Inputs: Window name; Field name; input value
Output: <<Win>>
Description: updates any window text object

set text
Inputs: <<Window>>; Item; Data input
Description: Updates a display item (text only) on the input window. Accepts a string or integer

update pc
Description: Deactivates PC dialog box, gets input value and updates PC value enables the menu bar

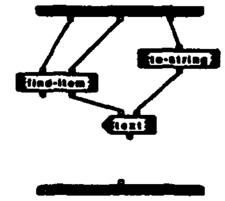
get string
Inputs: <<Window>>; Item name
Output: value of text

get pc window
Description: Activates PC dialog Box, and disables menu bar

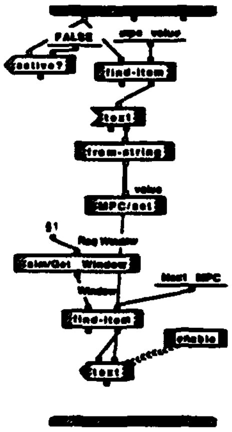


sim/set text 1:2

sim/set text 2:2

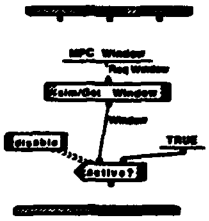


sim/Set MPC 1:1

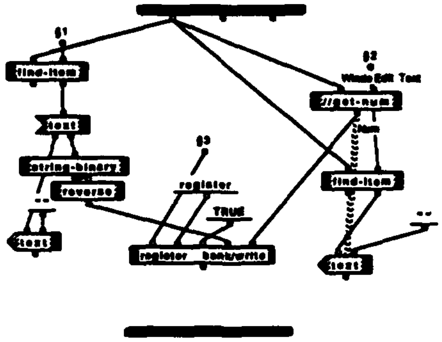


§1 Micro Simulator

sim/get mpc 1:1

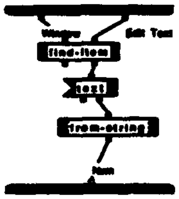


sim/initialize register 1:1

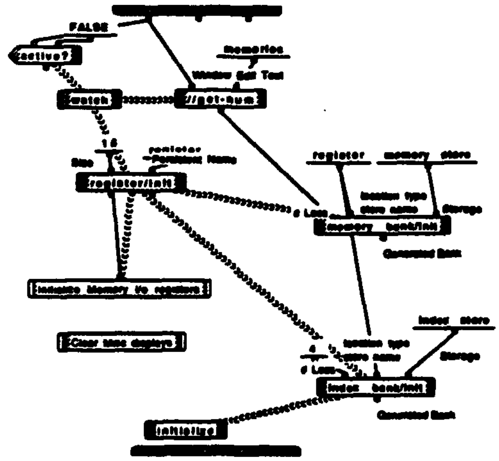


$1 register value
$2 register number
$3 register store

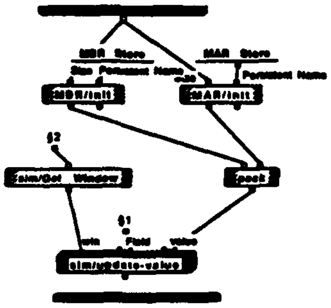
sim/get-num 1:1



sim/initial values 1:1

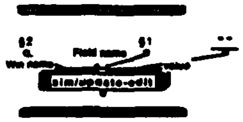


■ sim/initial values 1:1 ■ Initialize Memory I/o registers 1:1



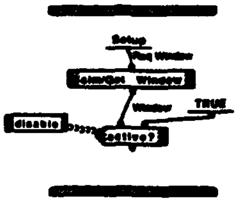
$1 ("MBR text" "MAR text")
$2 Micro Simulator

■ sim/initial values 1:1 ■ Clear Misc displays 1:1

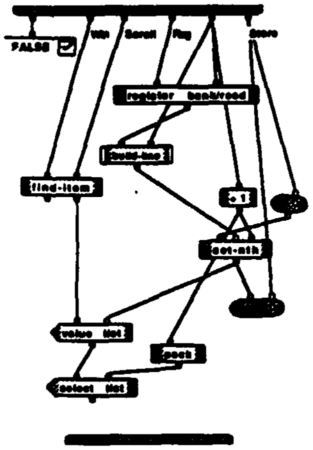


$1 ("MIR text" "Bus1 text" "Bus2 text" "ALU text")
$2 Micro Simulator

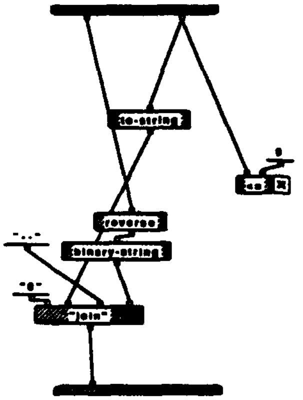
■ sim/define 1:1



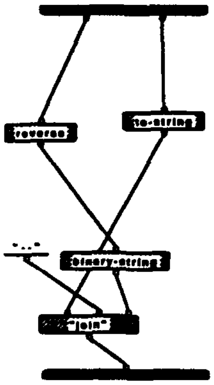
sim/update-scroll 1:1



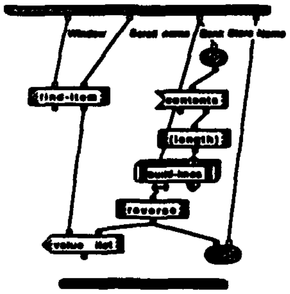
sim/update-scroll 1:1 build-line 1:2



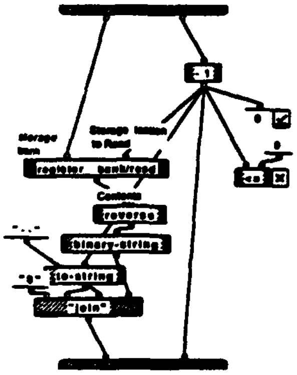
sim/update-scroll 1:1 build-lines 2:2



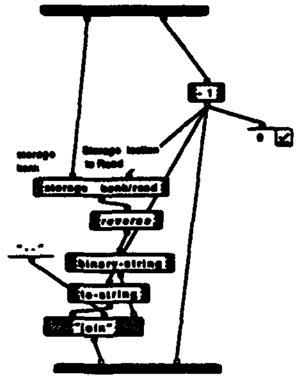
sim/initial-scroll 1:1



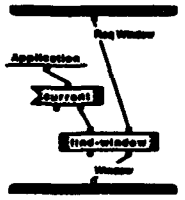
sim/initial-scroll 1:1 build-lines 1:2



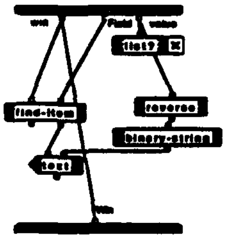
sim/initial-scroll 1:1 build-lines 2:2



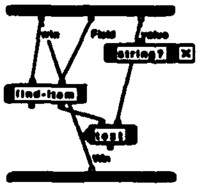
sim/Get Window 1:1



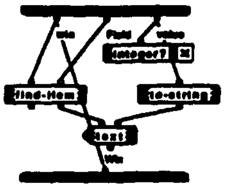
sim/update-value 1:4



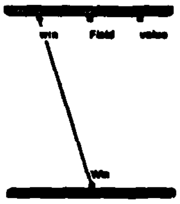
sim/update-value 2:4



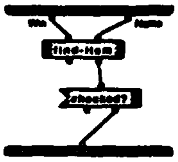
sim/update-value 3:4



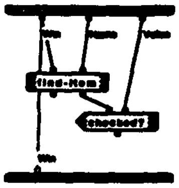
sim/update-value 4:4



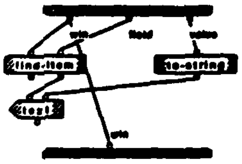
sim/setting 1:1



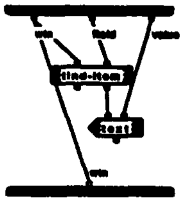
sim/set-setting 1:1



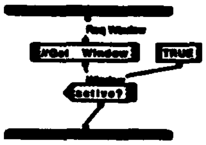
sim/update-integer 1:1



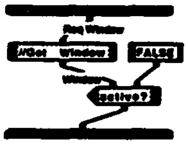
sim/update-string 1:1



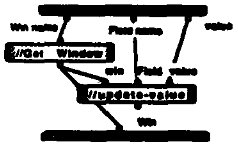
sim/activate 1:1



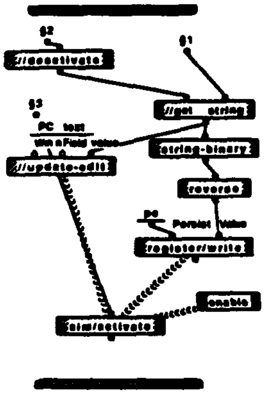
sim/deactivate 1:1



sim/update-edit 1:1

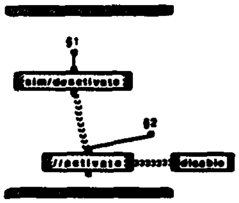


sim/update pc 1:1



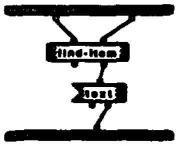
§1 Register Input
§2 Change Register
§3 Micro Simulator

sim/get pc window 1:1



§1 Micro Simulator
§2 Change Register

sim/get string 1:1



▽ index bank

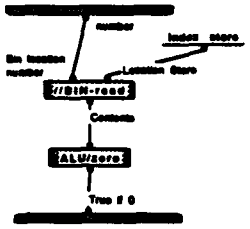
contents

index bank

zero index

Inputs: index number
Outputs: Boolean (T/F)
Description: Outputs status of selected index register if zero.

index bank/zero index 1:1



▽ IR

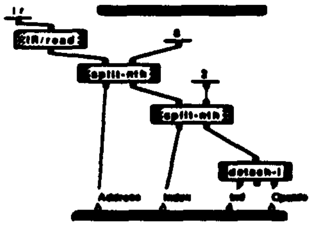
( )
▽
contents

IR

parse

Outputs: Address; Index; Ind; Opcode
Description: uses ir store to output results

IR/parse 1:1



▽ micro control

micro control

mcu
Description: reads control store and processes type one microinstruction

read control store
Outputs: Contents of Control Store
Description: Based on MPC value, passes contents of control store location

Bus1 Data
Description: Reads Control Store & generates Bus 1 Data based on control store location contents

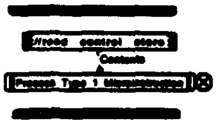
Bus2 Data
Description: Reads Control Store & generates Bus 2 data based on control store location contents

Bus3 Signals
Inputs: Alu result; PSR
Description: generates bus 3 data based on control store contents.

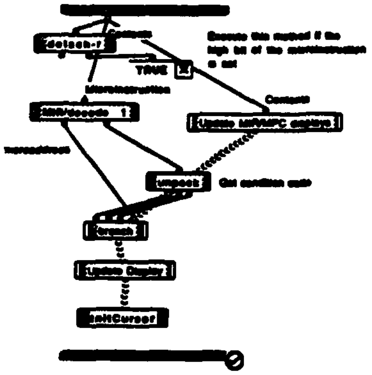
ALU Signals
Outputs: TRA1; TRA2; ADD; COMP; SHL; SHR
Description: Generates ALU Control signals.

Memory Signals
Outputs: Read; Write (Booleans)
Description: Generates R/W control Signals

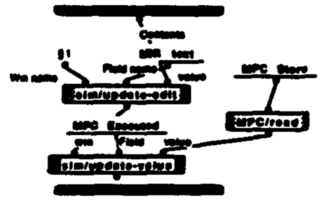
micro control/mcu 1:1

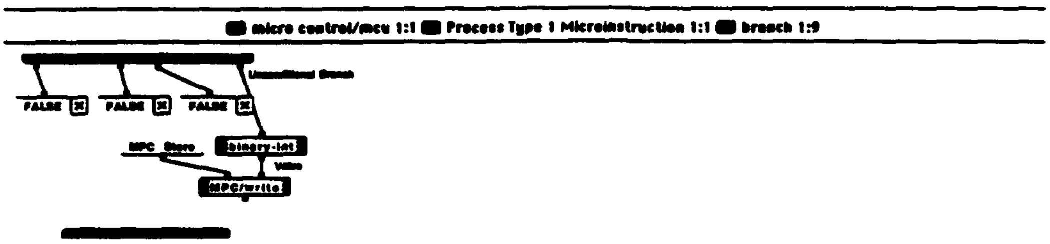


micro control/mcu 1:1 Process Type 1 Microinstruction 1:1

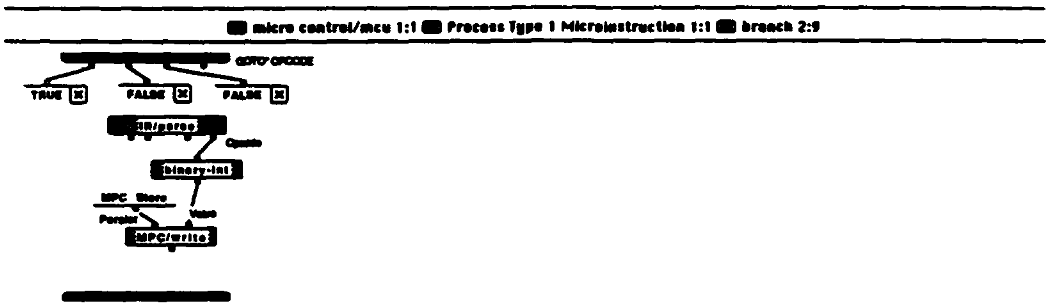


micro control/mcu 1:1 Process Type 1 Microinstruction 1:1 Update MIR/MPC displays 1:1

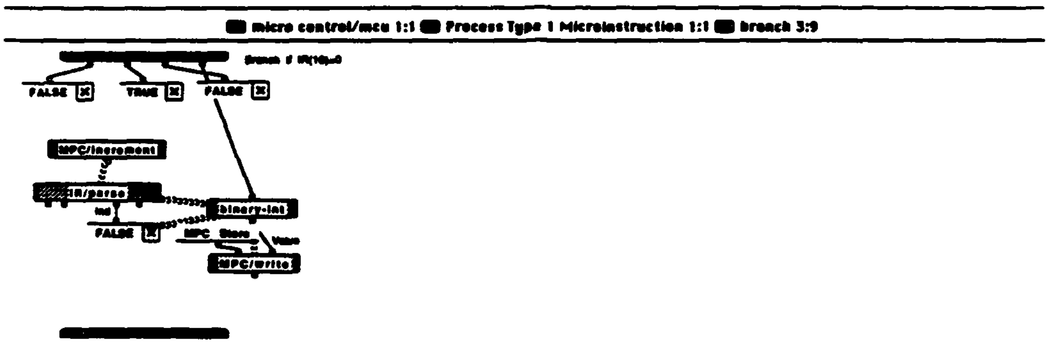




micro control/mcu 1:1 Process Type 1 Microinstruction 1:1 branch 1:9

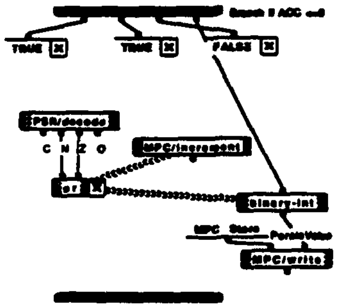


micro control/mcu 1:1 Process Type 1 Microinstruction 1:1 branch 2:9

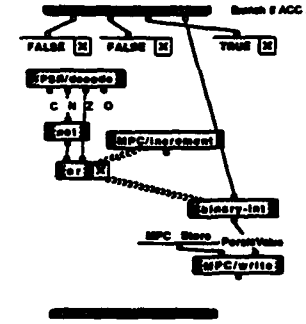


micro control/mcu 1:1 Process Type 1 Microinstruction 1:1 branch 3:9

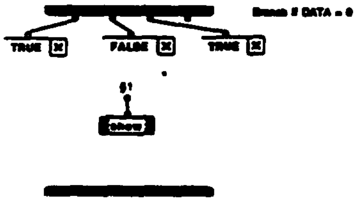
micro control/mcu 1:1 Process Type 1 Microinstruction 1:1 branch 4:9



micro control/mcu 1:1 Process Type 1 Microinstruction 1:1 branch 5:9

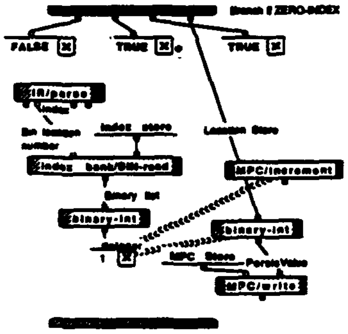


micro control/mcu 1:1 Process Type 1 Microinstruction 1:1 branch 6:9

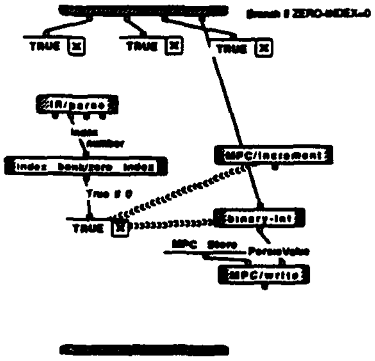


§1 Microcode caused Branch if DATA=0...this is bad

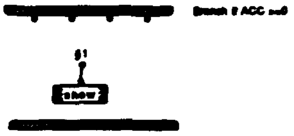
micro control/mcu 1:1 Process Type 1 Microinstruction 1:1 branch 7:9



micro control/mcu 1:1 Process Type 1 Microinstruction 1:1 branch 8:9

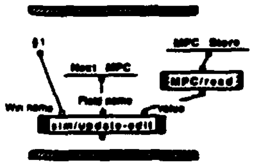


micro control/mcu 1:1 Process Type 1 Microinstruction 1:1 branch 9:9



;1 Branch local inside store controller fall to default case

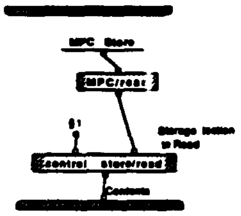
micro control/mcu 1:1 Process Type 1 Microinstruction 1:1 Update Display 1:1



micro control/mcu 1:1 Process Type 1 Microinstruction 1:1 Update Display 1:1

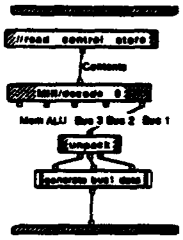
§1 Micro Simulator

micro control/read control store 1:1

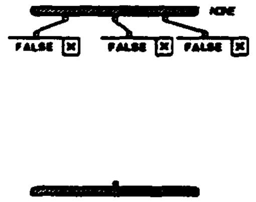


§1 control store

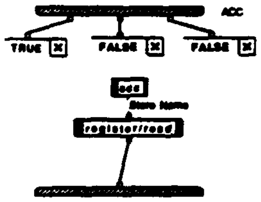
micro control/Bus1 Data 1:1



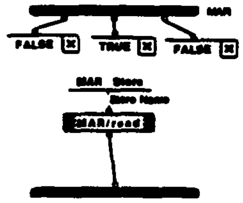
micro control/Bus1 Data 1:1 generate bus1 data 1:8



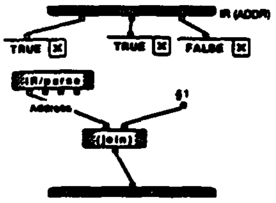
micro control/Bus1 Data 1:1 generate bus1 data 2:8



micro control/Bus1 Data 1:1 generate bus1 data 3:8

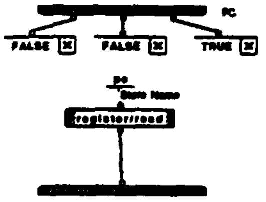


micro control/Bus1 Data 1:1 generate bus1 data 4:8

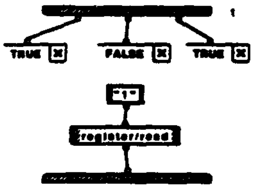


$1 (FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE)

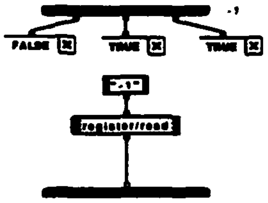
micro control/Bus1 Data 1:1 generate bus1 data 5:8



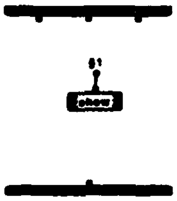
micro control/Bus1 Data 1:1 generate bus1 data 6:8



micro control/Bus1 Data 1:1 generate bus1 data 7:8

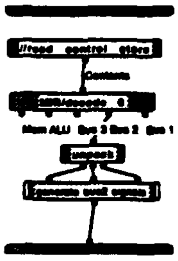


micro control/Bus1 Data 1:1 generate bus1 data 8:8

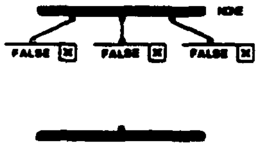


§1 Error in generate bus1 data

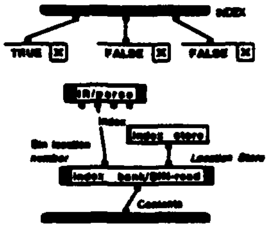
micro control/Bus2 Data 1:1



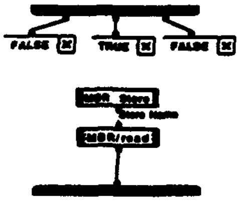
micro control/Bus2 Data 1:1 generate bus2 signals 1:5



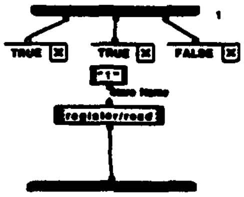
micro control/Bus2 Data 1:1 generate bus2 signals 2:5



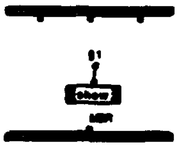
micro control/Bus2 Data 1:1 generate bus2 signals 3:5



micro control/Bus2 Data 1:1 generate bus2 signals 4:5

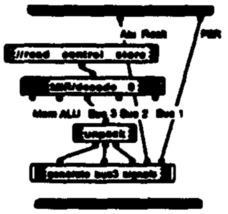


micro control/Bus2 Data 1:1 generate bus2 signals 5:5

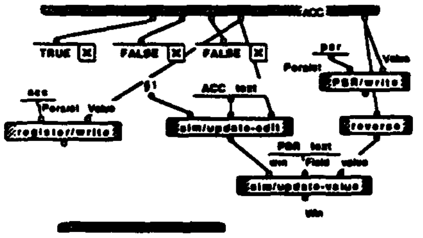


§1 Error in generate bus2 signals case

micro control/Bus3 Signals 1:1

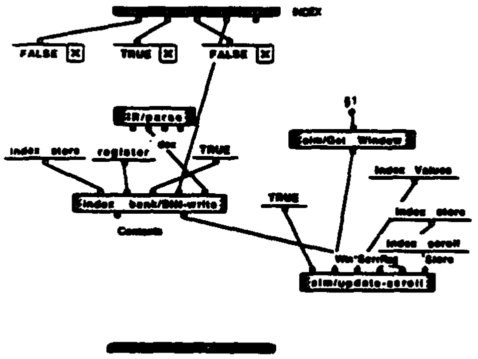


micro control/Bus3 Signals 1:1 generate bus3 signals 1:8

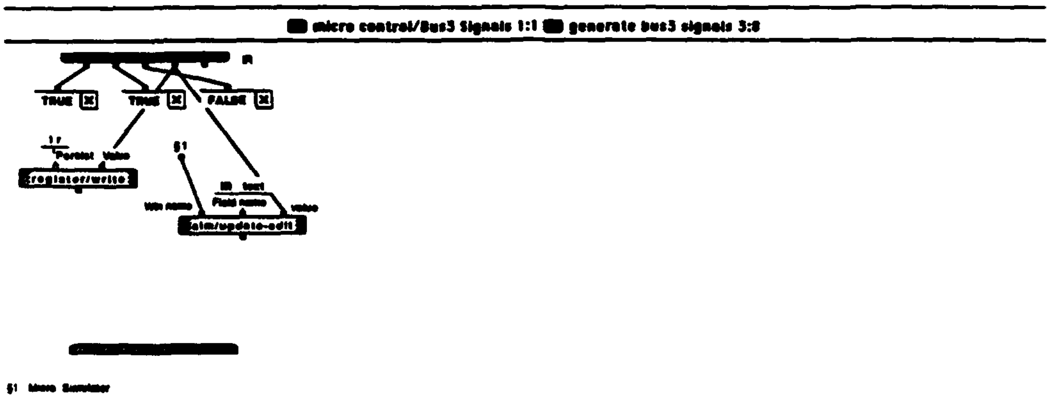


§1 Micro Simulator

micro control/Bus3 Signals 1:1 generate bus3 signals 2:8

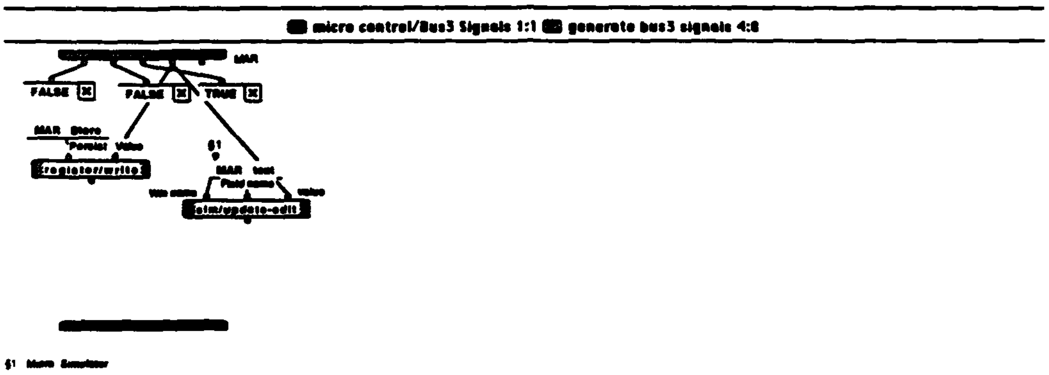


§1 Micro Simulator



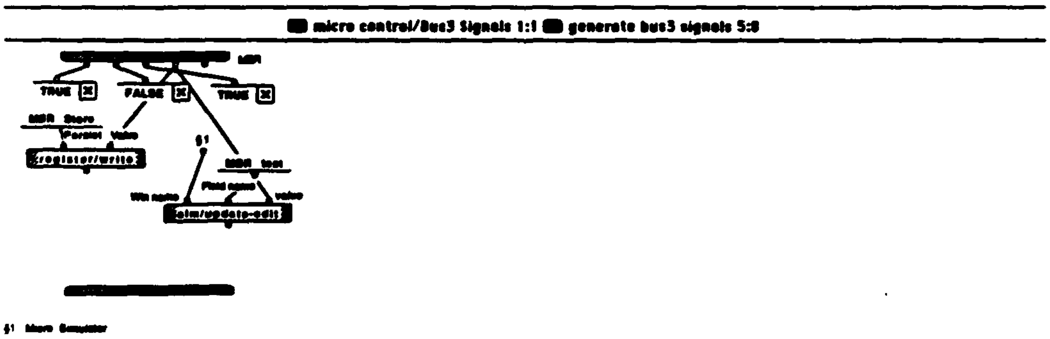
micro control/Bus3 Signals 1:1 generate bus3 signals 3:8

§1 Micro Simulator



micro control/Bus3 Signals 1:1 generate bus3 signals 4:8

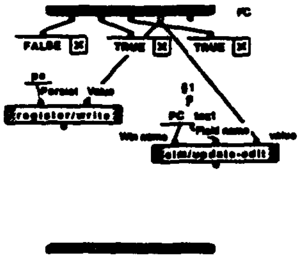
§1 Micro Simulator



micro control/Bus3 Signals 1:1 generate bus3 signals 5:8

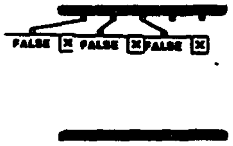
§1 Micro Simulator

■ micro control/Bus3 Signals 1:1 ■ generate bus3 signals 6:8

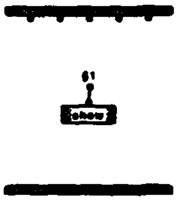


§1 Micro Simulator

■ micro control/Bus3 Signals 1:1 ■ generate bus3 signals 7:8

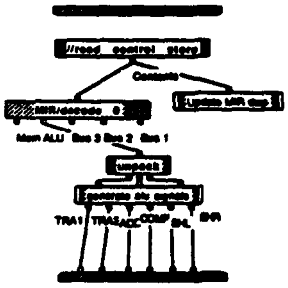


■ micro control/Bus3 Signals 1:1 ■ generate bus3 signals 8:8

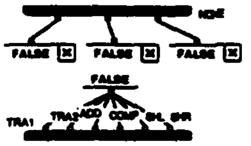


§1 Error in generate bus3 signals

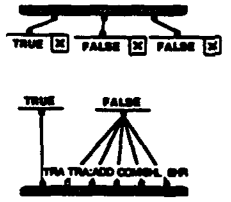
■ micro control/ALU Signals 1:1



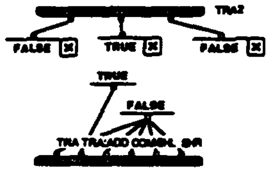
■ micro control/ALU Signals 1:1 ■ generate alu signals 1:8



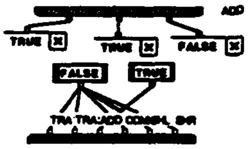
■ micro control/ALU Signals 1:1 ■ generate alu signals 2:8



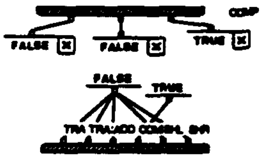
■ micro control/ALU Signals 1:1 ■ generate alu signals 3:8



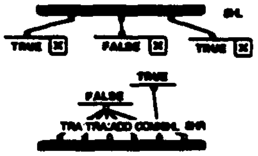
■ micro control/ALU Signals 1:1 ■ generate alu signals 4:8



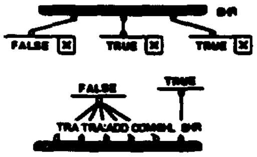
■ micro control/ALU Signals 1:1 ■ generate alu signals 5:8



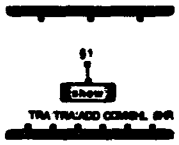
micro control/ALU Signals 1:1 generate alu signals 6:8



micro control/ALU Signals 1:1 generate alu signals 7:8

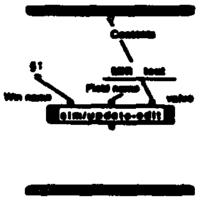


micro control/ALU Signals 1:1 generate alu signals 8:8



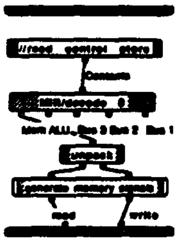
§1 Error in generate alu signals local method

micro control/ALU Signals 1:1 Update MIR disp 1:1

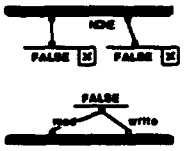


§1 Micro Simulator

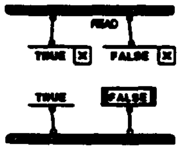
micro control/Memory Signals 1:1



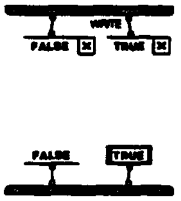
■ micro control/Memory Signals 1:1 ■ generate memory signals 1:4



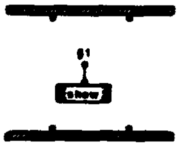
■ micro control/Memory Signals 1:1 ■ generate memory signals 2:4



■ micro control/Memory Signals 1:1 ■ generate memory signals 3:4



■ micro control/Memory Signals 1:1 ■ generate memory signals 4:4



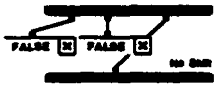
§1 Error in generate memory signals local method?

▽ shifter

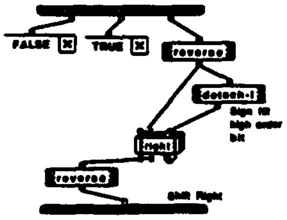
■ shifter

shifter Inputs: two control signals and boolean list.
Outputs:result (left, right, nodiff)

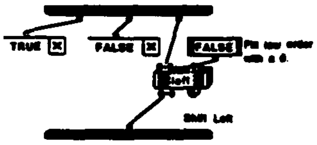
■ shifter/shifter 1:4



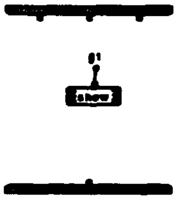
■ shifter/shifter 2:4



■ shifter/shifter 3:4

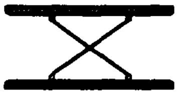


■ shifter/shifter 4:4

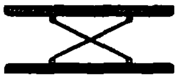


§1 Error in ALU/shifter

■ shifter/shifter 2:4 ■ right 1:1



■ shifter/shifter 3:4 ■ left 1:1



▽ ALU

ALU

bit not — Input: Bin list
output: not of Bin list

bit add — Input: Two bin lists
output: sum of bin list

math — Inputs: ADD; COMP; SHR; SHL; TRA1;
TRA2; BUS1; BUS2
Outputs: ALU result; PSR

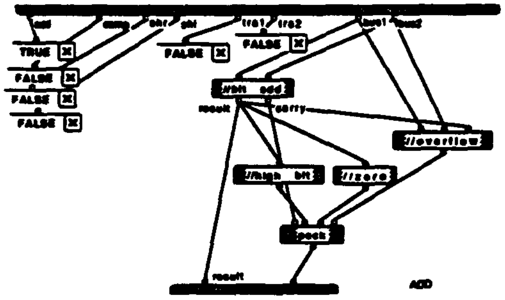
zero — Input: boolean lst
output: True if zero
false if not zero

high bit — Input: list of boolean
output: high order bit

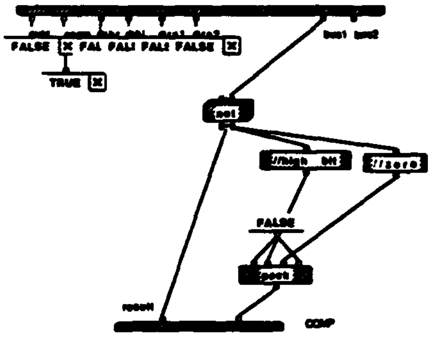
overflow — Inputs: Three boolean lists
Outputs: Boolean (T/F)

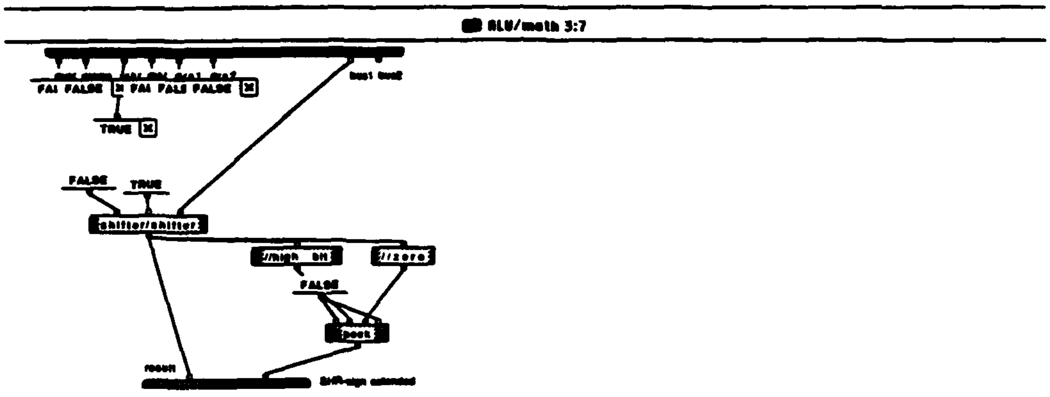
bit and — Input: two Bin lists
output: 1 bin list (and of A & B)

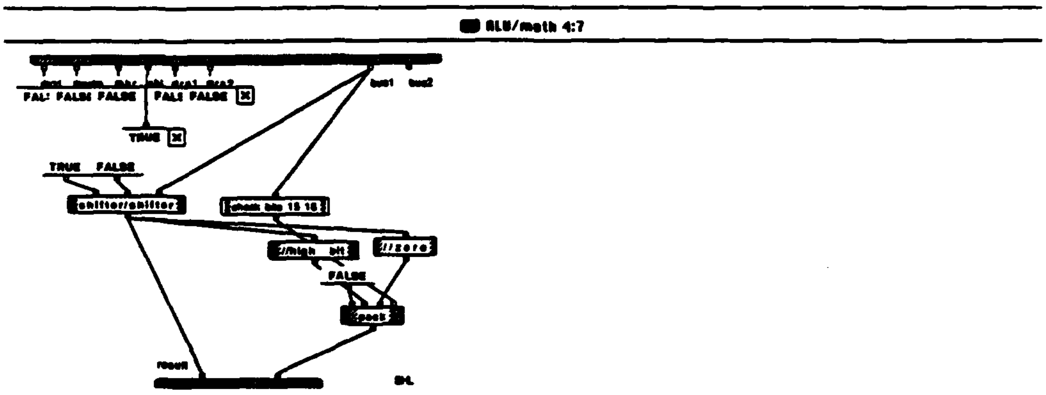
ALU/math 1:7

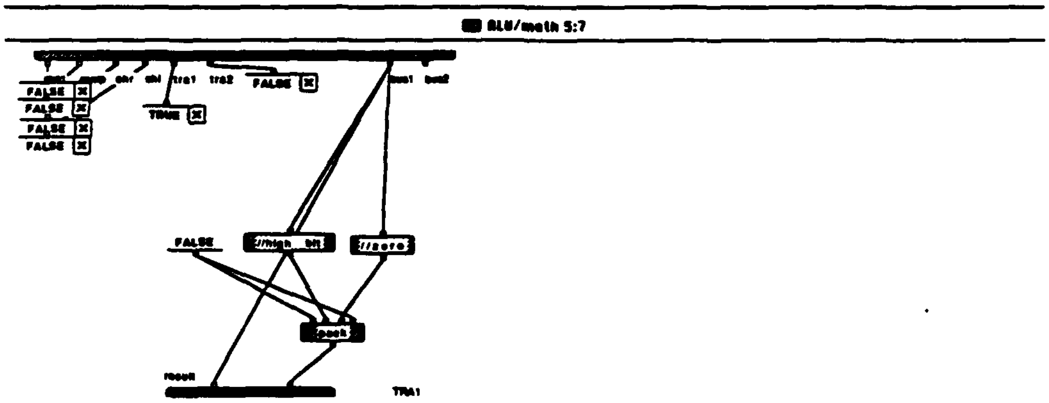


ALU/math 2:7

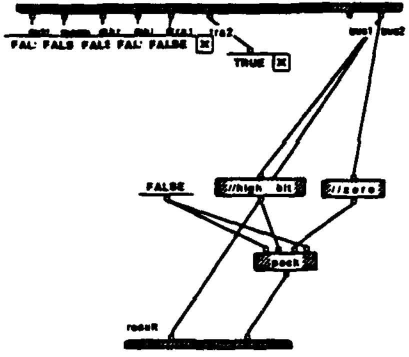



ALU/math 3:7
bus1 bus2
FAl FALSE FAl FALS FALSE ☒
TRUE ☒
FALSE
TRUE
shifter/shifter
//high bit
//zero
FALSE
pack
result
Shift-sign extended

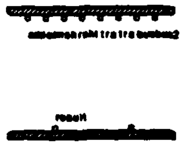

ALU/math 4:7
bus1 bus2
FAL FALSE FALSE FALS FALSE ☒
TRUE ☒
TRUE FALSE
shifter/shifter
check bits 15 15
//high bit
//zero
FALSE
pack
result
SHL


ALU/math 5:7
shr shl tra1 tra2
FALSE ☒
bus1 bus2
FALSE ☒
FALSE ☒
FALSE ☒
FALSE ☒
TRUE ☒
FALSE
//high bit
//zero
pack
result
TRA1

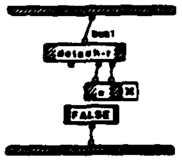
▨ ALU/math 6:7



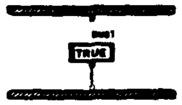
▨ ALU/math 7:7



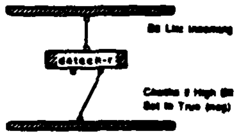
▨ ALU/math 4:7 ▨ check bits 15 16 1:2



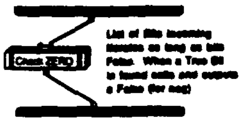
▨ ALU/math 4:7 ▨ check bits 15 16 2:2



▨ ALU/high bit 1:1

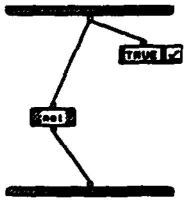


ALU/zero 1:1

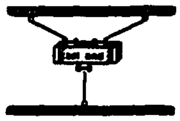


List of Bits incoming iterates as long as bits False. When a True Bit is found exits and outputs a False (for neg)

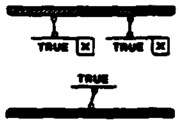
ALU/zero 1:1 Check ZERO 1:1



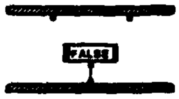
ALU/bit and 1:1



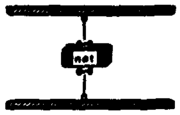
ALU/bit and 1:1 Bit and 1:2



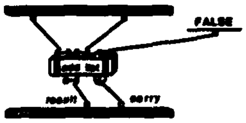
ALU/bit and 1:1 Bit and 2:2



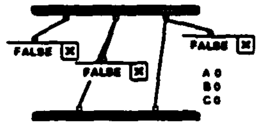
ALU/bit not 1:1



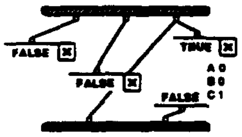
ALU/bit add 1:1



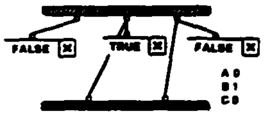
ALU/bit add 1:1 add list 1:9



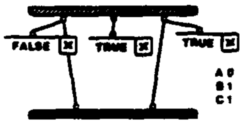
ALU/bit add 1:1 add list 2:9



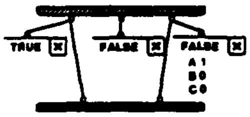
ALU/bit add 1:1 add list 3:9



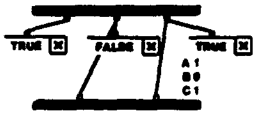
ALU/bit add 1:1 add list 4:9



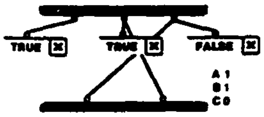
ALU/bit add 1:1 add list 5:9



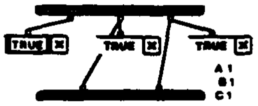
ALU/bit add 1:1 add list 6:9



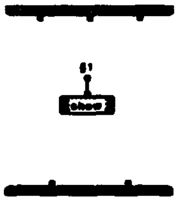
ALU/bit add 1:1 add list 7:9



ALU/bit add 1:1 add list 8:9

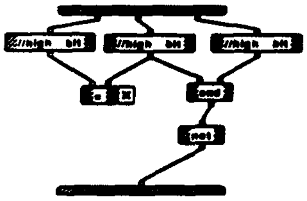


ALU/bit add 1:1 add list 9:9

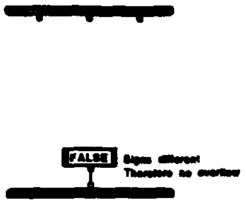


§1 error in add list

ALU/overflow 1:2



ALU/overflow 2:2



▽ control store

control store

init: Inputs: Store Name
Description: Generates empty instance and loads into a persistent

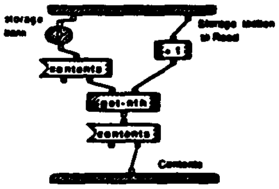
load: Inputs: Store Name
Description: Loads text file representing micro code into control store

read: Inputs: Store Name; Int Location
Outputs: Contents

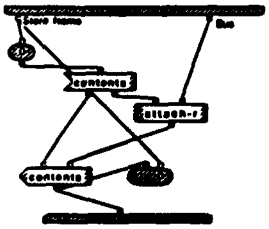
add location: Inputs: Store Name; Bus
Outputs: Revised Control Store Contents
Description: Adds a new Microinstruction to the Control Store contents.

menu load: Inputs: None
Invokes loading of control store with appropriate persistent name

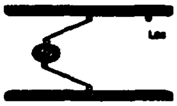
control store/read 1:1



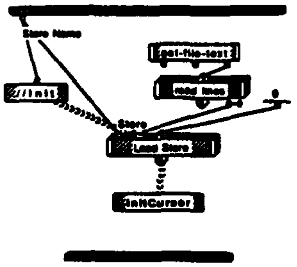
control store/add location 1:2



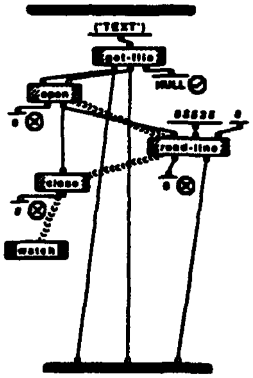
control store/add location 2:2



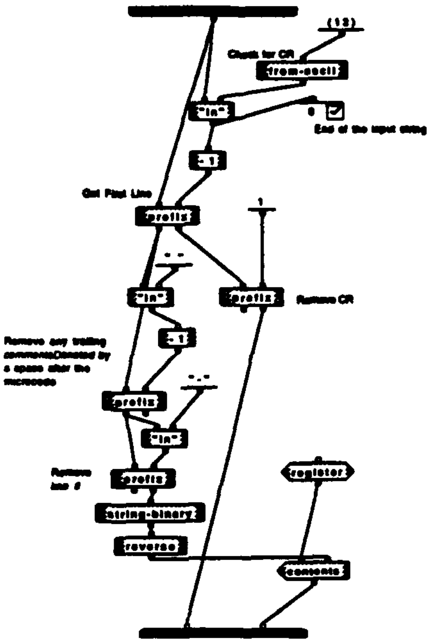
control store/load 1:1



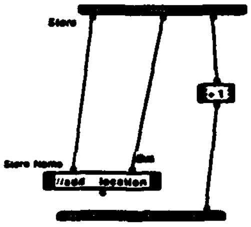
control store/load 1:1 get-file-text 1:1



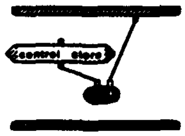
control store/load 1:1 read lines 1:1



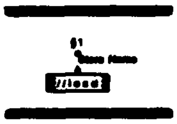
control store/load 1:1 Load Store 1:1



control store/init 1:1



control store/menu load 1:1



§1 control store

∇ storage bank

storage bank

init — Inputs: #Locs; Location type; Storage params
Outputs: Generated Bank

read — Inputs: Storage Bank; Location #
Outputs: Location contents

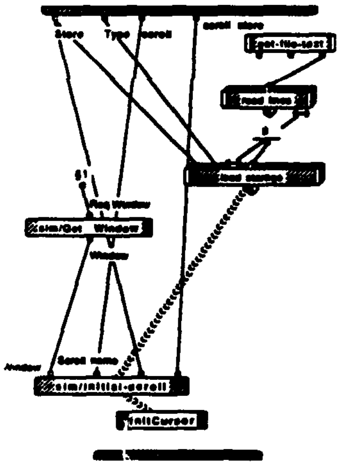
load — Inputs: Store; Type; Scroll; Scroll Store
Outputs: None
Description: Loads text file into Store and initializes associated scroll

write — Inputs: Store Name; Location Type, Control; Bus; Loc
Outputs: Contents

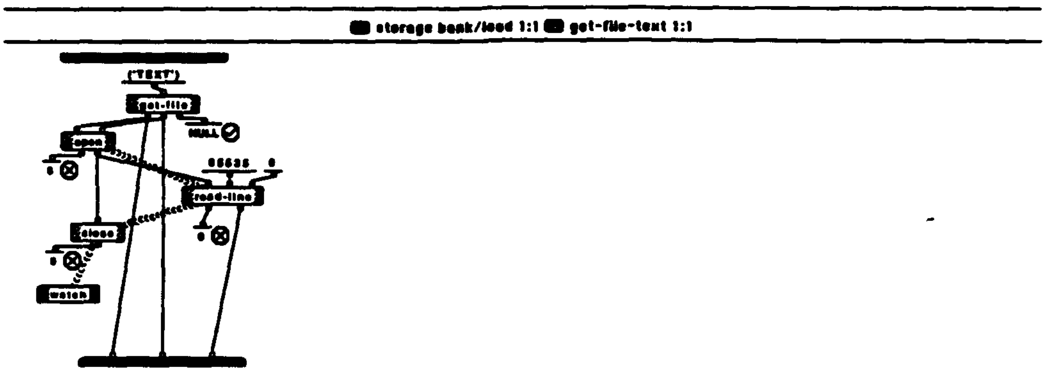
BIN-read — Inputs: Binary Location; Location Store
Outputs: Contents

BIN-write — Inputs: Store Name; Location Type; Control; Bus; Loc
Outputs: Contents, Integer Location

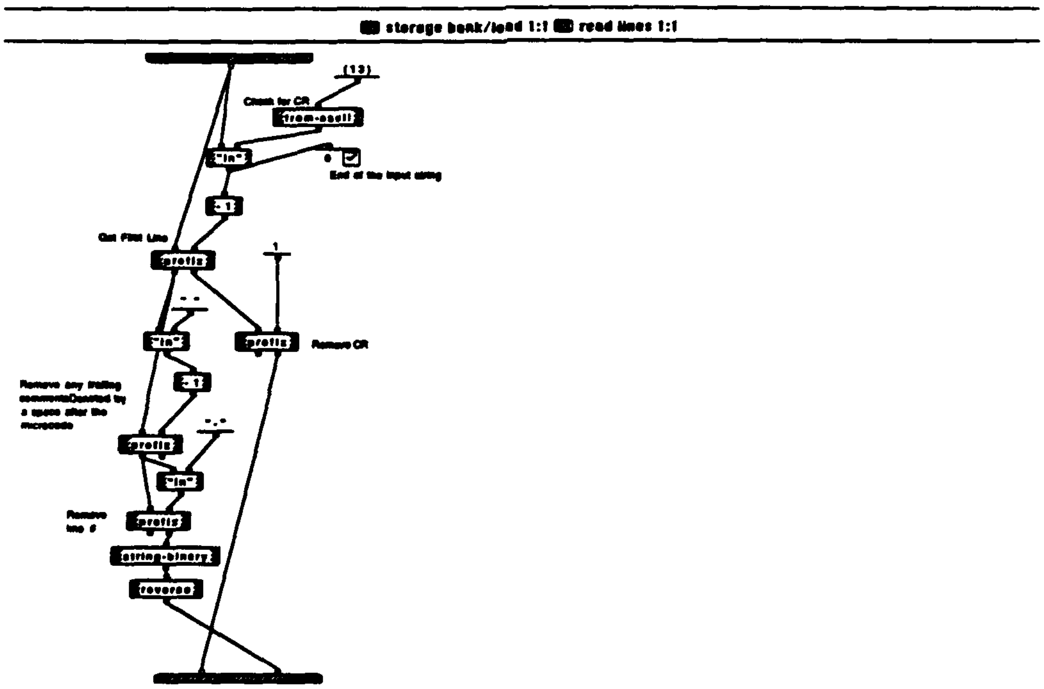
storage bank/load 1:1



[1] Micro Simulator

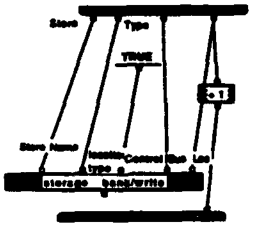


storage bank/load 1:1 get-file-text 1:1

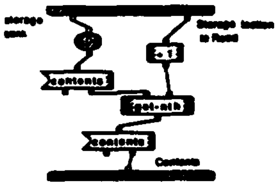


storage bank/load 1:1 read lines 1:1

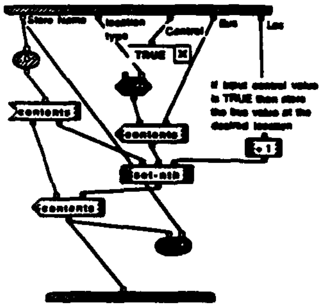
storage bank/load 1:1 load storage 1:1



storage bank/read 1:1

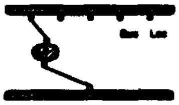


storage bank/write 1:2

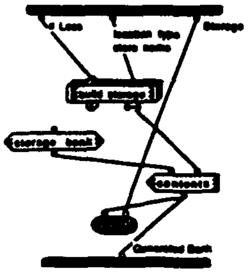


If input control value is TRUE then store the bus value at the desired location

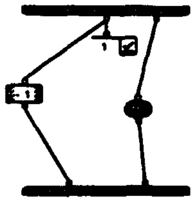
storage bank/write 2:2



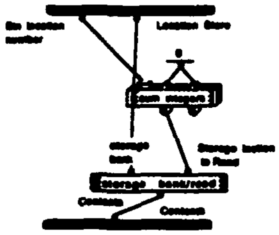
storage bank/init 1:1



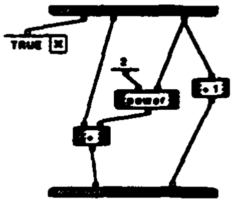
storage bank/init 1:1 build storage 1:1



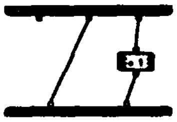
storage bank/BIN-read 1:1



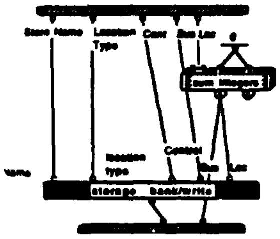
storage bank/BIN-read 1:1 sum integers 1:2



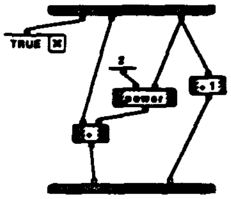
■ storage bank/BIN-read 1:1 ■ sum integers 2:2



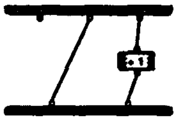
■ storage bank/BIN-write 1:1



■ storage bank/BIN-write 1:1 ■ sum integers 1:2



■ storage bank/BIN-write 1:1 ■ sum integers 2:2



▽ memory bank

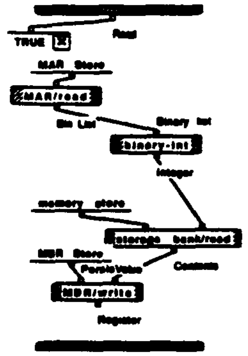
memory bank

load — Description: supplies appropriate inputs to superclass's load method for loading the memory bank.

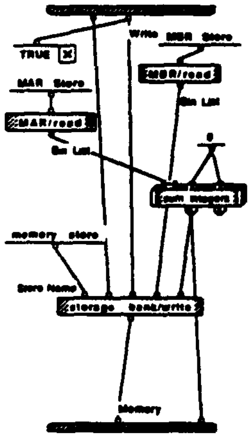
write — Inputs: Bus Input; Write Control (T/F)
Outputs: Location Value; Int Location

read — Inputs: Control (T/F) to enable read
Description: Uses MAR contents as location to read memory. Writes results to MBR

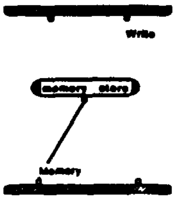
memory bank/read 1:1



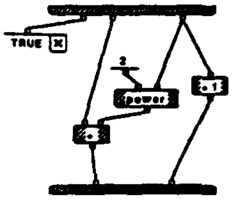
memory bank/write 1:2



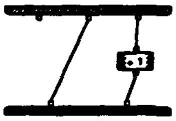
memory bank/write 2:2



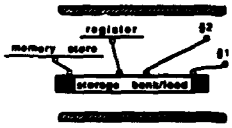
memory bank/write 1:2 sum integers 1:2



memory bank/write 1:2 sum integers 2:2



memory bank/load 1:1



§1 Memory Scroll
§2 Memory Values

▽ register bank

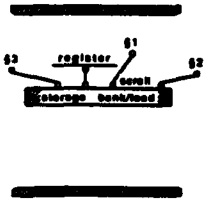
contents

register bank

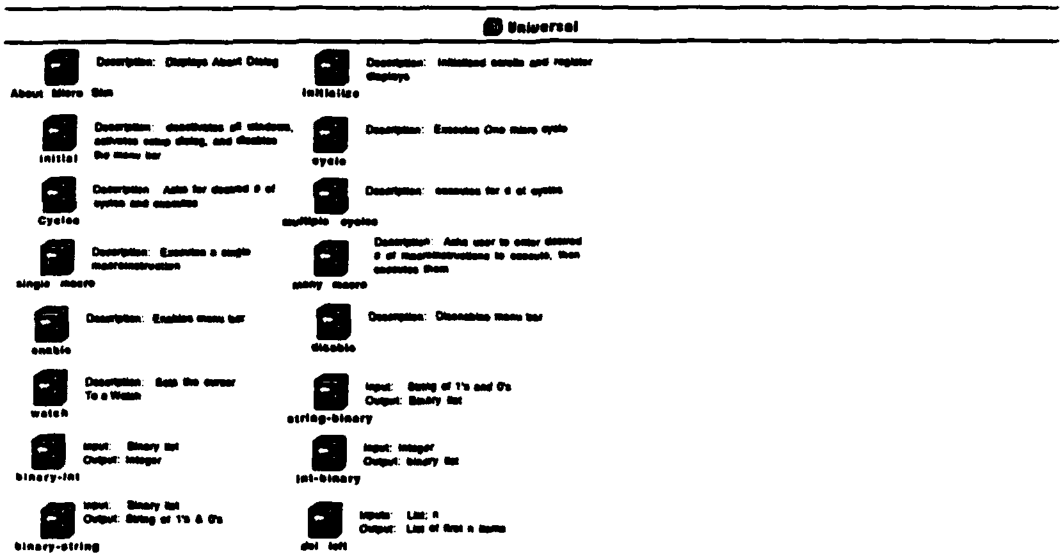
load

Description: supplies appropriate inputs to superclass's load method for loading the register bank

register bank/load 1:1



§1 Register Values
§2 Register Scroll
§3 register store



Universal

About Micro Sim — Description: Displays About Dialog

Initialize — Description: Initializes scrolls and register displays

initial — Description: deactivates all windows, activates setup dialog, and disables the menu bar

cycle — Description: Executes One micro cycle

Cycles — Description: Asks for desired # of cycles and executes

multiple cycles — Description: executes for # of cycles

single macro — Description: Executes a single macroinstruction

many macro — Description: Asks user to enter desired # of macroinstructions to execute, then executes them

enable — Description: Enables menu bar

disable — Description: Disenables menu bar

watch — Description: Sets the cursor To a Watch

string-binary — Input: String of 1's and 0's Output: Binary list

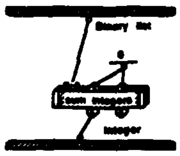
binary-int — Input: Binary list Output: Integer

int-binary — Input: Integer Output: binary list

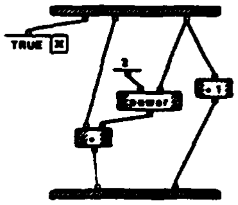
binary-string — Input: Binary list Output: String of 1's & 0's

del left — Inputs: List; n Output: List of first n items

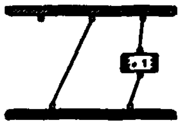
binary-int 1:1



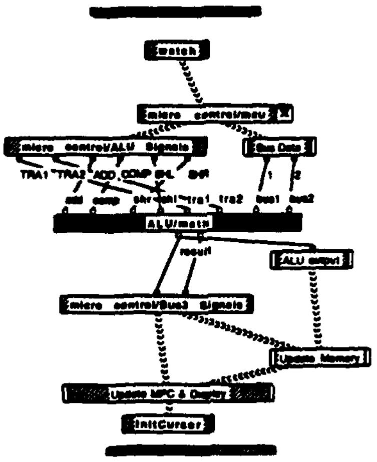
binary-int 1:1 sum integers 1:2



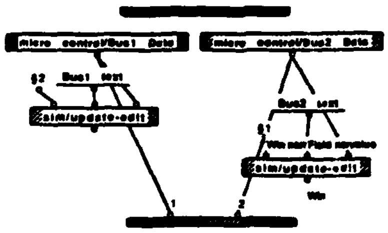
binary-int 1:1 sum integers 2:2



cycle 1:1

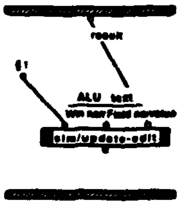


cycle 1:1 Bus Data 1:1



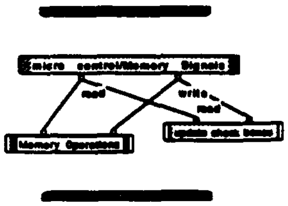
§1 Micro Simulator
§2 Micro Simulator

cycle 1:1 ALU output 1:1

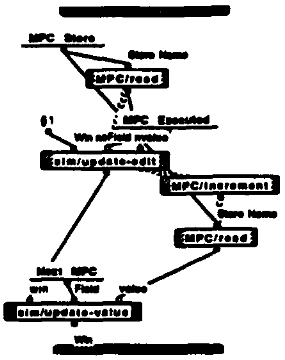


§1 Micro Simulator

cycle 1:1 Update Memory 1:1

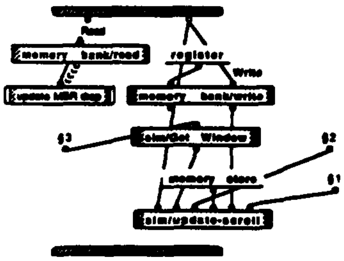


cycle 1:1 Update MPC & Display 1:1



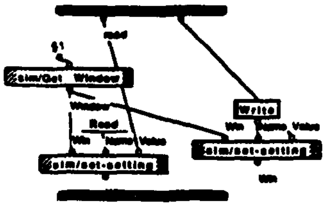
§1 Micro Simulator

cycle 1:1 Update Memory 1:1 Memory Operations 1:1



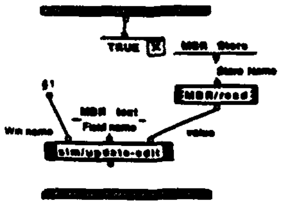
§1 Memory Scroll
§2 Memory Values
§3 Micro Simulator

cycle 1:1 Update Memory 1:1 update check boxes 1:1



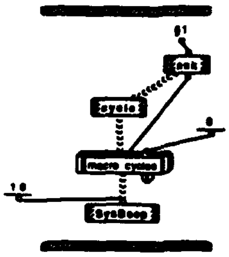
§1 Micro Simulator

cycle 1:1 Update Memory 1:1 Memory Operations 1:1 update MBR disp 1:1



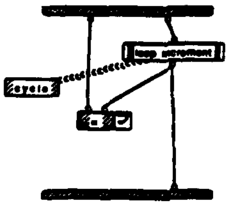
§1 Micro Simulator

many macro 1:1

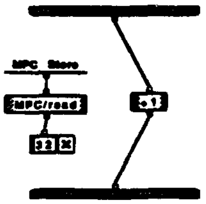


§1 Enter desired number of Macro Instructions to execute:

many macro 1:1 macro cycles 1:1

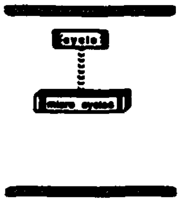


■ many macro 1:1 ■ macro cycles 1:1 ■ loop increment 1:2

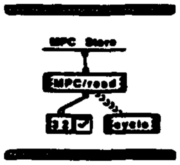


■ many macro 1:1 ■ macro cycles 1:1 ■ loop increment 2:2

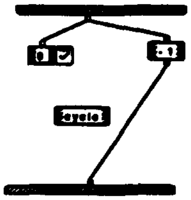
■ single macro 1:1



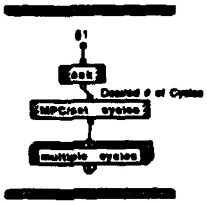
■ single macro 1:1 ■ micro cycles 1:1



■ multiple cycles 1:1

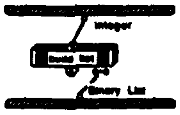


Cycles 1:1

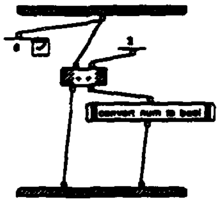


§1 Enter Desired # of Machine Cycles to Run

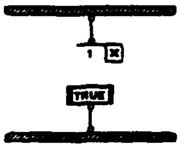
int-binary 1:1



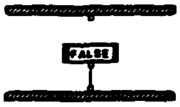
int-binary 1:1 build list 1:1



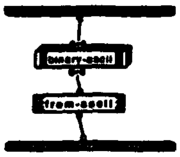
int-binary 1:1 build list 1:1 convert num to bool 1:2



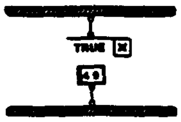
int-binary 1:1 build list 1:1 convert num to bool 2:2



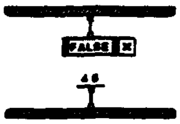
■ binary-string 1:1



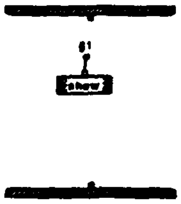
■ binary-string 1:1 ■ binary-ascii 1:3



■ binary-string 1:1 ■ binary-ascii 2:3

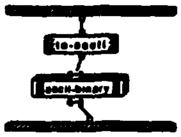


■ binary-string 1:1 ■ binary-ascii 3:3

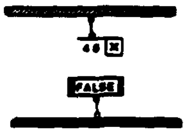


§1 error in binary-ascii

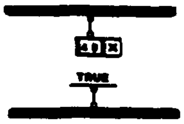
■ string-binary 1:1



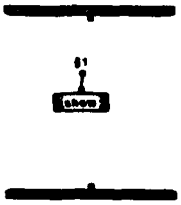
■ string-binary 1:1 ■ ascii-binary 1:3



string-binary 1:1 ascii-binary 2:3

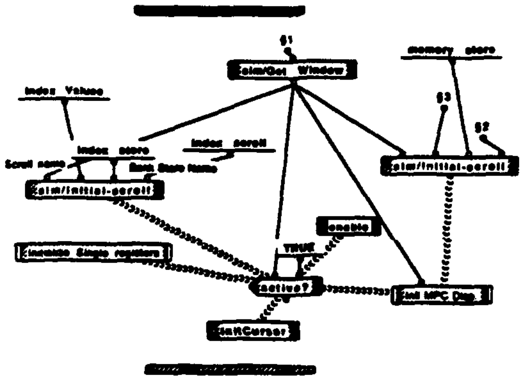


string-binary 1:1 ascii-binary 3:3



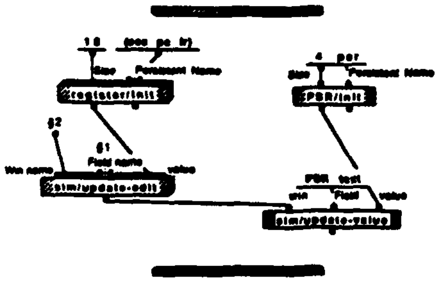
§1. error in ascii-binary

initialize 1:1



§1 Macro Simulator
§2 Memory Scroll
§3 Memory Values

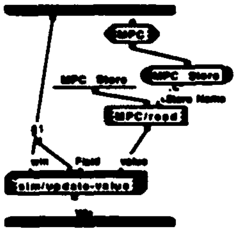
initialize 1:1 initialize single registers 1:1



initialize 1:1 initialize Single registers 1:1

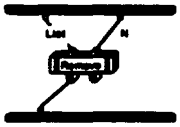
§1 ("ACC test" "PC test" "IR test")
§2 Micro Simulator

initialize 1:1 init MPC Disp. 1:1

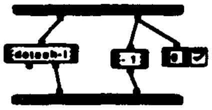


§1 ("MPC Executed" "Next MPC")

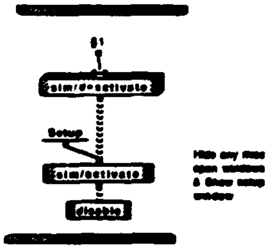
del left 1:1



del left 1:1 Remove 1:1

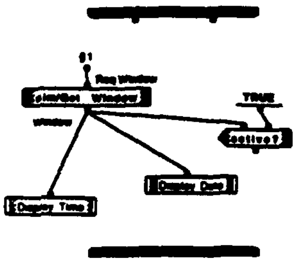


initial 1:1



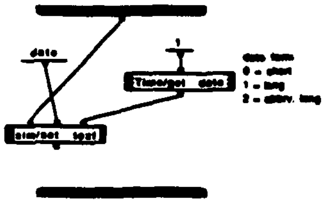
§1 ("Micro Simulator" "Setup" "MPC Window" "About MPG Micro Simulator")

About Micro Sim 1:1

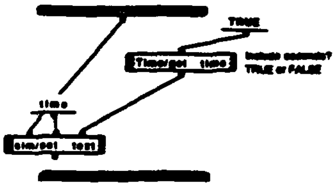


§1 About NPS Micro Simulator

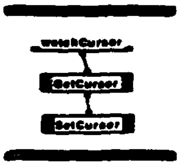
About Micro Sim 1:1 Display Date 1:1



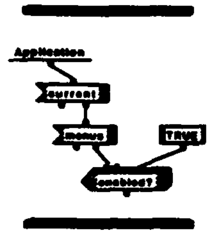
About Micro Sim 1:1 Display Time 1:1



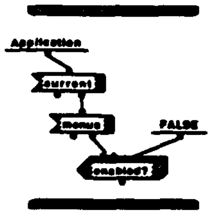
watch 1:1



■ enable 1:1



■ disable 1:1



LIST OF REFERENCES

Apple Computer, Inc., "Inside Macintosh Volume I." Addison-Wesley Publishing Company, Inc. Reading, MA, 1985.

Cox, P. T. and Pietrzykowski, T. "Prograph: a Pictorial View of Object-Oriented Programming." Technical Report 8902, The Gunakara Sun Systems, Ltd., 1989.

de Paula, E. G. and Nelson, M. L. "Designing a Class Hierarchy." Proceedings of the Technology of Object-Oriented Languages and Systems International Conference 5 (Tools USA 1991), Santa Barbara, CA, July 1991, pp. 203-218.

Frei, M. "Simulating von-Neumann Machines in an Object Oriented Environment." IEE Colloquium, November 1989, pp. 5/1-5/6.

Micallef, J. "Encapsulation, Reusability and Extensibility in Object-Oriented Programming Languages." Journal of Object-Oriented Programming, Vol. 1, No. 1, April/May 1988. pp. 12-35.

Mulcare, D., "Object-Based Discrete-Event Simulation of Concurrent Real-Time System Architectures." Proceedings of the 1990 Summer Computer Simulation Conference, July 1990, pp. 184-190.

Nelson, M. L. "An Introduction to Object-Oriented Programming." Technical Report NPS52-90-024, Naval Postgraduate School, Monterey, CA, April 1990.

Papazoglou, M., Pawlak, A., Wrona, W. "Multiprocessor Modelling: An Example of Object-Oriented Development." Microprocessing and Microprogramming, 25, 1989, pp. 213-219.

Shiva, S. G. "Computer Design & Architecture." HarperCollins, New York, NY, 1991.

Stefik, M., and Bobrow, D. "Object-Oriented Programming: Themes and Variations." The AI Magazine, Vol 6, No. 4, Winter 1986, pp. 40-62.

Sugimoto, A., Abe, S., Kuroda, M., and Katou, S., "An Object-Oriented approach for Interactive Microprogram Simulator." Systems and Computer in Japan, Vol. 19, No. 1, 1988, pp. 47-57.

Tanenbaum, A. S. "Structured Computer Organization." Prentice-Hall, Englewood Cliffs, NJ, 1984.

The Gunakara Sun Systems "Prograph Reference Manual." The Gunakara Sun Systems, Ltd., Halifax, Nova Scotia (Canada), July 1990.

Tomek, I., "Simulation of Computer Architecture." Mini and Microcomputers and their Applications. Proceedings of the ISMM International Symposium, pp. 493-495, June 1985 pp. 493-495.

Wegner, P. "Dimensions of Object-Based Language Design." Special Issue of SIGPLAN Notices; Vol 22, No. 12, Dec 1987 pp. 168-182.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	LT Kevin A. Fontes 4811 S. Hunt Road Gustine, CA 95322	2
4.	MAJ Michael L. Nelson, Code CS/Ne Naval Postgraduate School Monterey, CA 93943-5100	5
5.	Amr Zaky, Code CS/Za Naval Postgraduate School Monterey, CA 93943-5100	1
6.	Robert B. McGhee, Code CS Naval Postgraduate School Monterey, CA 93943-5100	1
7.	CDR Thomas J. Hoskins, Code 37 Naval Postgraduate School Monterey, CA 93943-5100	1
8.	Lou Stevens, Code CS/St Naval Postgraduate School Monterey, CA 93943-5100	1